Quixote User's Manual

Quixote User's Manual

The Quixote User's Manual was prepared by the technical staff of Innovative Integration on June 25, 2008.

For further assistance contact:

Innovative Integration
2390-A Ward Ave
Simi Valley, California 93065

PH: (805) 578-4260
FAX: (805) 578-4225

email: techsprt@innovative-dsp.com
Website: www.innovative-dsp.com

This document is copyright 2008 by Innovative Integration. All rights are reserved.

VSS \ Distributions \ Quixote \ Documentation \ OO_Manual \ QuixoteMaster.odm

#XXXXXX

Rev 2.0

Table of Contents

Introduction....14

Real Time Solutions!....14
Vocabulary....14
What is Velocia? 14
What is Quixote? 14
What is Malibu? 15
What is C++ Builder?....15
What is Microsoft MSVC?....15
What kinds of applications are possible with Innovative Integration hardware?....15
Why do I need to use Malibu with my Baseboard?....15
Finding detailed information on Malibu....16
Online Help....16
Innovative Integration Technical Support....16
Innovative Integration Web Site....16
Typographic Conventions....17

Installation....18

Host Hardware Requirements....18
Software Installation....18
Tools Registration....21
Bus Master Memory Reservation Applet....21
Hardware Installation....22
JTAG Emulator Hardware Installation (for DSP boards Only)....22
PCI Pod-Based Emulator Installation....22
Baseboard Installation....23
A Few Considerations BEFORE Power-up....23
It cannot be overemphasized: 23
After Power-up....24
Code Composer Studio Setup....24
Borland Builder Setup and Use....26
Automatic saving of project files and forms during debugging....27
Static-binding of built executables. 27
Appropriate library and include paths. 28

Quixote Baseboard Overview....29

Quixote Hardware Overview....29
Quixote Block Diagram....30
Baseboard Enumeration....30
Multiple Baseboards 31
PCI Interrupt Configuration and Compatibility....31
Quixote Logics....31
Quixote Host Software Overview....32
The Innovative::Quixote object....32
Initialization....32
Logic Configuration....32
FPA Configuration....33
Target to Host Communication....33

The Innovative::C64xDsp object....33
Initialization....33
COFF File Downloading....33
Target to Host Communication....34
Quixote Target Software Overview....34
Communication to the Host....34
Analog Input Stream Devices....35
Analog Output Stream Devices....35
Timebase Objects....35
Timebase Software Objects....36
Digital Io....37
Quixote Architecture....38
Processing Core....39
Analog I/O....40
Timing and Synchronization Features....40
PCI Bus....41
Digital Signal Processor....41
DSP External Memory....42
DSP Initialization....42
DSP JTAG Debugger Support....43
FPGA JTAG Support....44
Host/Target Communications....45
Overview....45
CPU Busmastering Interface....46
CPU Busmastering Implementation....46
Packet Based Transfers....46
Blocking Interface....46
Maximum Transfer Size....46
Malibu Library Host Support for CPU Busmastering....47
Packet Notification Events....47
Target (Pismo Library) Support for CPU Busmastering....47
Packetized Message Interface....48
Message Mailbox Emulation....48
The Message System....48
Host-side Message Objects....49
Target Side Message Objects....50
Message Communication....51
C++ Terminal I/O....52
Target Software....52
Tutorial....52
Programming with the Pismo Class Library....54
Simple To Use....54
Not Just for C++ Experts....55
Unique Feature Support for each Baseboard....55
DSP Programming on the Baseboard....55
Device Drivers....55
Advantages of using DSP/BIOS drivers....56
How to use a DSP/BIOS driver....56
Driver-specific control functions....57

Driver Buffer Model....58
Driver Types....59
Driver Implementation....59
DMA-enabled Drivers....60
Simplified Use....60
Multitasking Friendly....61
Analog Timebase Objects....61
Timebase Usage....61
Interrupt Handling....62
Interrupts in a C++ Environment....62
The Pismo Solution....62
Class Irq....63
Interrupt Lock Classes....63
Interrupt Binder Templates....64
Class InterruptHandler....64
Class ClassMemberHandler Template....64
Class FunctionHandler Template....65
EDMA and QDMA Handling....65
Class DmaSettings....66
Class Qdma....66
Class Edma....67
Linked and Chained blocks....69
Class EdmaMaster....70
Quixote Example Programs....70
The Next Step: Developing Custom Code....71

Building a Target DSP Project....72

Writing a Program....74
Host Tools for Target Application Development....74
Components of Target Code (.cpp, .cdb, .cmd, .pjt)....75
Edit-Compile-Test Cycle using Code Composer Studio....75
Automatic projectfile creation....75
Rebuilding a Project....75
IIMain replaces main....76
Running the Target Executable....76
Note:....76
Anatomy of a Target Program....77
Use of Library Code....78
Example Programs....78
The Next Step: Developing Custom Code....79

Writing Custom Applications....80

The Asnap Example....80
Tools Required....80
Program Design....81
The Host Application....81
User Interface....81
Host Side Program Organization....82
ApplicationIo....82
Initialization....82
Logic Loading....84

COFF File Loading....85
Starting a Data Capture....86
The Target Application....87
Main Thread....87
The Capture Thread....87
Configuring Data Acquisition....89
The Close() Method....91
Data Logging....91
Command Packet Reception....91
Data Packet Reception....92

Developing Host Applications....93

Borland Turbo C++....93
Microsoft Visual Studio 2005....95

Applets....97

Common Applets....97
Registration Utility (NewUser.exe)....97
ReserveMemoryDsp....98
....98
Data Analysis Applets....98
Binary File Viewer Utility (BinView.exe)....98
Target Programming Applets....99
Target Project Copy Utility (CcsProjectCopy.exe)....99
Demangle Utility (Demangle.exe)....99
COFF Section Dump Utility (CoffDump.exe)....99
JTAG Diagnostic Utility (JtagDiag.exe)....100
DEF Conversion Utility (DefConvert.exe)....100
PCI Configuration Utility (Config.exe)....101
RtdxTerminal - Terminal Emulator....101
Important Note:....102
Terminal Emulator Menu Commands....102
The File Menu....102
The DSP Menu....102
The Form Menu....103
The Help Menu....103
Options Tab:....103
Display Group....104
Sounds Group....105
Coff Load Group....105
Debugger Group....106
Terminal Emulator Command Line Switches....106
Applets for the C64x DSP Processor....106
DSP Detector (C64xDetect.exe)....106
COFF Downloader (C64xDownload.exe)....107
C64x DSP EEProm Configuration Utility (C64xEeprom.exe)....107
Applets for the Quixote Baseboard....107
Board Finder (Finder.exe)....107
Target Number....108
Blink....108
On/OFF....108

PCI Logic Update Utility (VsProm.exe)....108
User Logic Download Utility (QuixoteLogicLoader.exe)....108

Custom Logic Development....109

Overview....109
Target Devices....110
Code Development Tools....110
Steps for Successful Use....110
Using the FPGA Framework....111
Developing FPGA Firmware....114
Steps to Follow for Logic Builds....115
Logic Files and Hierarchy....116
Fitting Results for the Framework Logic....116
Adding functionality to the Framework Logic....118
Logic Design Methods and Code Development....118
DAC Data Path....118
A/D Data Path....119
DSP Interface Logic....120
DSP EMIF A Data Read Interface....123
DSP EMIF Write Interface....125
EMIF B Interface....126
A/D and D/A FIFOs and Controls....126
SBSRAM Memory....129
Digital I/O from the Virtex2....129
Clock Domains....129
Clocks for Use in the Logic....130
Pin Banks....131
PMC J4 Test Port....131
Constraints....131
Timing Constraints....132
IOB Constraints....133
IO Standard Constraints....133
Pin Constraints....133
Simulation....133
Required Software and Hardware....134
Setting Up the Simulation....134
Loading the Testbench....134
Running the Simulations....134
Modifying the Simulations....135
Some Things to Watch Out For....137
Loading the Logic Image....138

Quixote Analog Input and Output....139

Analog Inputs....139
Overview....139
A/D Anti-alias Filtering and Input Circuitry....140
A/D Triggering and Data Collection....142
Enabling the data acquisition engine....142
A/D Error Correction....142
A/D Calibration....143
A/D Data Flow Controls and Data Buffering....143

Monitoring A/D Data Flow Errors....144
Analog Input Timebases and Triggering....145
Software Support....145
Timebase Software Objects....147
Hardware Support....148
A/D Clock Sources....148
Start and Stop Trigger Selections....148
Threshold Triggering....150
A/D Frame Timer....151
A/D Frame Counter....152
Single-Shot Mode....152
Pretrigger Mode....152
Analog Input Comparator Trigger....153
Analog Outputs....153
Smoothing Filter Characteristics....155
D/A Triggering and Data Collection....156
Enabling The D/A data conversion engine....156
D/A Data Format....157
D/A Error Correction....157
D/A Calibration....158
D/A Data Flow Controls and Data Buffering....158
Monitoring D/A Data Flow Errors....159
Analog Output Timebases and Triggering....159
Software Support....160
Timebase Software Objects....161
Hardware Support....162
D/A Clock Sources....162
Start and Stop Triggers....163
D/A Frame Timer....165
D/A Frame Counter....165
Single-Shot Mode....165

Target Peripheral Devices....166

DSP Interrupts....166
Software Support....166
Hardware Implementation....166
Conditioning for Interrupt Input Signals....167
Shared/ Dedicated Interrupts....167
External User Interrupt Usage....168
Digital I/O....168
Software Support....168
Hardware Implementation....169
Digital I/O Timing....170
Digital IO Electrical Characteristics....170
Timers....170
Software Support....171
Hardware Implementation....171
On-chip Timers....171
AD9852 Direct Digital Synthesizer....172
PXI Timing Support....173
Software Support....173

Hardware Implementation....173
External Clock Input....174
Alert Log....174
Overview....174
Software Support....175
Alert Objects....176
Alert Binder Templates....176
Using Alerts in Application Code....176
Host Side Support....177
Hardware Implementation....177
Alert Log Message Format....178
Controlling and Monitoring the Alert Log....179
Alert Log Data Flow....179
Serial EEPROM Interface....180
StarFabric....180
StarFabric Modes....180
Hardware Interface....180
Software Support....180
PMC Module Support (not released)....181
PMC PCI Interface....181
PMC J4 Support....181

Connector Pinouts and Physical Information....182

Analog IO Connectors....182
JP5 – UD Digital I/O Connector....182
JN4 - PMC Private IO Connector (Rev B or higher only)....185
JP2 – Virtex2 JTAG Connector....186
JP7 – DSP JTAG Connector....186
Board Layout and Connectors....187
cQuixote Revision A Connectors....188
cQuixote Revision B Connectors....189
cQuixote Revision C and Revision D Connectors....190
cQuixote Revision E Connectors....191

List of Tables

Table 1. Quixote Feature Matrix....29
Table 2. Quixote Initialization Methods....32
Table 3. IUsesVirtexFpgaLoader Configuration Methods....32
Table 4. C64xDsp Initialization Methods....33
Table 5. IUsesOnboardCpu Loading Methods....34
Table 6. Data Acquisition Modes for the Quixote....35
Table 7. Quixote Timebase Objects....36
Table 8. Quixote ‘C6416 DSP EMIF Control Register Initialization Values....42
Table 9. TIIMessage Header Field Access....49
Table 10. TIIMessage Data Section Interface....50
Table 11. IIMessage Header Field Access....50
Table 12. IIMessage Data Section Interface....51
Table 13. Message Sending and Receiving Methods....51
Table 14. Timebase Operations....61
Table 15. Interrupt Lock Classes....63
Table 16. Quixote Example Programs....70
Table 17. Pismo Example Programs....78
Table 18. Development Tools for the Asnap Example....80
Table 19. Logic Devices on Quixote Variants....110
Table 20. Quixote Memory Map....112
Table 21. Dsp Memory Space Assignments....123
Table 22. Available Clocks for The Logic....130
Table 23. A/D channel ordering and data set format....142
Table 24. A/D Calibration memory locations and format....143
Table 25. Timebase Operation Modes....146
Table 26. Analog In Timebase Object Type....147
Table 27. A/D conversion timebase (0x64130000)....148
Table 28. A/D start trigger selection register (0x80100000)....149
Table 29. A/D start trigger selection register (0x80100000)....149
Table 30. Threshold Channel and Hysteresis Register (0x801A0000)....151
Table 31. Threshold value register (0x80180000)....151
Table 32. A/D Frame Timer Register (0x801C0000)....151
Table 33. A/D Frame Counter Register (0x801E0000)....152
Table 34. D/A channel ordering and data set format....157
Table 35. D/A calibration memory registers and format....157
Table 36. D/A FIFO Level and Status Register Field Description....158
Table 37. Timebase Operation Modes....160
Table 38. Analog Out Timebase Object Types....161
Table 39. D/A conversion timebase (0x64130000)....163
Table 40. D/A start trigger selection register (0x80200000)....163
Table 41. D/A stop trigger selection register (0x80240000)....164
Table 42. D/A Trigger polarity and type selection register (start = 0x80220000 stop = 0x80260000)....164
Table 43. D/A Frame Timer Register (0x802C0000)....165
Table 44. D/A Frame Counter Register (0x802E0000)....165
Table 45. External Interrupt Input Control Register Addresses....166
Table 46. External Interrupt Input Control Register Definitions....166
Table 47. Interrupt polarity and Type selection (polarity = 0x64110000, type = 0x64100000)....167

Table 48. External Interrupt Status and Acknowledge Register Addresses....168
Table 49. DigitalIo Class Operations....169
Table 50. UD Digital I/O Configuration Control Register Definition (0x80300000)....169
Table 51. Digital IO Port Addresses....169
Table 52. Digital I/O Port Timing Parameters....170
Table 53. Timer Management Objects....171
Table 54. DSP On-chip Timer Clock Source Control Register (0x64140000)....171
Table 55. DDS Control Register (0x68010000)....172
Table 56. PXI Control Register Definition (0x80120000)....173
Table 57. PXI Signals (when master)....174
Table 58. Alert Log Enable Register (0x80060000)....177
Table 59. Alert Message Format (0x805E0000)....178
Table 60. Alert Log Event Types....179
Table 61. Analog IO Connectors (Rev B)....182
Table 62. UD Digital I/O Connector Pinout....182
Table 63. PMC JN4 Pinout for Rev B....185
Table 64. PMC JN4 Pinout for Rev C and Higher....185
Table 65. Virtex2 JTAG Connector Pinouts....186
Table 66. JP7 DSP JTAG Connector Pinouts....186

List of Figures

Figure 1. Block Diagram for the Quixote Baseboard....30
Figure 2. Quixote Block Diagram....38
Figure 3. Quixote Computing Core....39
Figure 4. Simplified Quixote A/D channel....40
Figure 5. Simplified Quixote DAC channel....40
Figure 6. Simplified View of Quixote PCI Bus Architecture....41
Figure 7. Quixote DSP JTAG Chain....44
Figure 8. Quixote FPGA JTAG Chain....44
Figure 9. Messaging System Objects....49
Figure 10. RtdxTerminal Options....104
Figure 11. Overview of FPGA External Connections....111
Figure 12. Logic Structure and Files....116
Figure 13. Map Results....117
Figure 14. PAR Results....118
Figure 15. DAC Interface Block Diagram....119
Figure 16. A/D Interface Block Diagram....120
Figure 17. DSP Asynchronous Timing. (Courtesy of Texas Instruments)....121
Figure 18. DSP Synchronous Read Timing. (Courtesy of Texas Instruments)....122
Figure 19. DSP Synchronous Write Timing. (Courtesy of Texas Instruments)....122
Figure 20. Logic Component Source Code....123
Figure 21. EMIF A Data Bus Read Interface Diagram....124
Figure 22. EMIF A Incoming Data Bus Registers....126
Figure 23. FIFO Basic Interface....126
Figure 24. FIFO Component Instantiation....127
Figure 25. Using The FIFO Interrupt Filter....128
Figure 26. A/D FIFO Read Control Signal....128
Figure 27. DAC FIFO Control Signal....128
Figure 28. Clock Domains in The Framework Logic....130
Figure 29. Clock Period Constraints....132
Figure 30. DSP Signals Timing Constraint....133
Figure 31. An IOB Register Constraint....133
Figure 32. IO Standard Constraint....133
Figure 33. A Pin Sample Pin Constraint....133
Figure 34. VHDL Procedure for Simulating a DSP Async Access....136
Figure 35. Simulating a DSP Async Access....136
Figure 36. VHDL Procedure for DSP Burst Read Access....137
Figure 37. Simulating DSP Interrupt Servicing....137
Figure 38. DCM Reset Code Fragment....138
Figure 39. Interconnections of the Analog Input Sub-System....140
Figure 40. Input Filter Circuitry....140
Figure 41. A/D anti-alias filter frequency response....141
Figure 42. Detail of A/D anti-alias filter frequency response....141
Figure 43. A/D FIFO Level and Status Register (0x805A0000)....143
Figure 44. A/D FIFO Interrupt Threshold Level Register (0x80020000)....144
Figure 45. A/D triggering fundamentals....145
Figure 46. A/D Trigger polarity and type selection register (start = 0x80120000, stop = 0x80160000)....150
Figure 47. Threshold trigger example with programmable hysteresis of 4 samples....151

Figure 48. Input Comparator Trigger....153
Figure 49. DAC Output Filtering....154
Figure 50. D/A Output Block Diagram....155
Figure 51. DAC smoothing filter frequency response....155
Figure 52. Detail of the DAC smoothing filter frequency response....156
Figure 53. D/A FIFO Level and Status Register (0x805C0000)....158
Figure 54. D/A FIFO Interrupt Threshold Level Register (0x800E0000)....159
Figure 55. DAC triggering Fundamentals....160
Figure 56. D/A Trigger Selection Logic....164
Figure 57. UD Digital I/O Configuration Control Register (0x80300000)....169
Figure 58. Digital I/O Port Timing....170
Figure 59. Digital I/O Connector Pin Orientation (JP5)....184
Figure 60. PMC JN4 Connector Pin out....185
Figure 61. cQuixote Rev. A Connectors....188
Figure 62. cQuixote Rev. B Connectors....189
Figure 63. cQuixote Rev. C and D Connectors....190
Figure 64. cQuixote Rev. E Connectors....191

Introduction

Real Time Solutions!

Thank you for choosing Innovative Integration, we appreciate your business! Since 1988, Innovative Integration has grown to become one of the world's leading suppliers of DSP and data acquisition solutions. Innovative offers a product portfolio unrivaled in its depth and its range of performance and I/O capabilities .

Whether you are seeking a simple DSP development platform or a complex, multiprocessor, multichannel data acquisition system, Innovative Integration has the solution. To enhance your productivity, our hardware products are supported by comprehensive software libraries and device drivers providing optimal performance and maximum portability.

Innovative Integration's products employ the latest digital signal processor technology thereby providing you the competitive edge so critical in today's global markets. Using our powerful data acquisition and DSP products allows you to incorporate leading-edge technology into your system without the risk normally associated with advanced product development. Your efforts are channeled into the area you know best ... your application.

Vocabulary

What is Velocia?
Velocia is an advanced architecture DSP baseboard that integrates a high performance Texas Instruments TMS320C64xx DSP and Xilinx high density programmable logic with high performance peripherals such as PMC modules, analog IO and interconnectivity interfaces. The powerful combination of the DSP and FPGA provide signal processing speed and flexibility for almost any DSP application. Each baseboard features a PCI backbone connecting the DSP, PMC, peripherals and StarFabric interfaces. The StarFabric interface (PICMG 2.17) provides unlimited and extremely flexible interconnection to other DSP cards, IO cards and host processor systems. Each Velocia card incorporates a high performance IO system with either on-board peripherals like A/D and D/As, or one or more PMC sites accommodating a wide range of I/O options.

What is Quixote?
Quixote is Innovative Integration’s Velocia-family baseboard for wireless, RADAR, ultrasound, high energy physics and other other demanding applications requiring speed and processing power. Quixote features a powerful processing core built

around Texas Instruments TMS320C6416 and Xilinx Virtex2 (2M or 8M gates) with 32MB of DSP RAM and 2MB of FPGA computation RAM (optional). Analog IO features include dual channels of 105 MHz A/D and D/A. A PMC site provides additonal IO expansion capability. System expansion is over StarFabric in a PICMG 2.17 compatible compact PCI chassis.

What is Malibu?
Malibu is the Innovative Integration-authored component suite, which combines with the Borland, Microsoft or GNU C++ compilers and IDEs to support programming of Innovative hardware products under Windows and Linux. Malibu supports both high-speed data streaming plus asynchronous mailbox communications between the DSP and the Host PC, plus a wealth of Host functions to visualize and post-process data received from or to be sent to the target DSP.

What is C++ Builder?
C++ Builder is a general-purpose code-authoring environment suitable for development of Windows applications of any type. Armada extends the Builder IDE through the addition of functional blocks (VCL components) specifically tailored to perform real-time data streaming functions.

What is Microsoft MSVC?
MSVC is a general-purpose code-authoring environment suitable for development of Windows applications of any type. Armada extends the MSVC IDE through the addition of dynamically created MSVC-compatible C++ classes specifically tailored to perform real-time data streaming functions.

What kinds of applications are possible with Innovative Integration hardware?
Data acquisition, data logging, stimulus-response and signal processing jobs are easily solved with Innovative Integration baseboards using the Malibu software. There are a wide selection of peripheral devices available in the Matador DSP product family, for all types of signals from DC to RF frequency applications, video or audio processing. Additionally, multiple Innovative Integration baseboards can be used for a large channel or mixed requirement systems and data acquisition cards from Innovative can be integrated with Innovative's other DSP or data acquisition baseboards for high-performance signal processing.

Why do I need to use Malibu with my Baseboard?
One of the biggest issues in using the personal computer for data collection, control, and communications applications is the relatively poor real-time performance associated with the system. Despite the high computational power of the PC, it cannot reliably respond to real-time events at rates much faster than a few hundred hertz. The PC is really best at processing data, not collecting it. In fact, most modern operating systems like Windows are simply not focused on real-time performance, but rather on ease of use and convenience. Word processing and spreadsheets are simply not high-performance real-time tasks.

The solution to this problem is to provide specialized hardware assistance responsible solely for real- time tasks. Much the same as a dedicated video subsystem is required for adequate display performance, dedicated hardware for real-time data

collection and signal processing is needed. This is precisely the focus of our baseboards – a high performance, state-of-the-art, dedicated digital signal processor coupled with real-time data I/O capable of flowing data via a 64-bit PCI bus interface.

The hardware is really only half the story. The other half is the Malibu software tool set which uses state of the art software techniques to bring our baseboards to life in the Windows environment. These software tools allow you to create applications for your baseboard that encompass the whole job - from high speed data acquisition, to the user interface.

Finding detailed information on Malibu
Information on Malibu is available in a variety of forms:

- Data Sheet (http://www.innovative-dsp.com/products/malibu.htm)
- On-line Help
- Innovative Integration Technical Support
- Innovative Integration Web Site (www.innovative-dsp.com)

Online Help

The online help system for Malibu is fully integrated into the excellent OpenHelp system provided with Builder. Help for Malibu is provided in a single file, Malibu.hlp which is installed beneath the main Builder C++ directory tree during the default installation. It provides detailed information about the components contained in Malibu - their Properties, Methods, Events, and usage examples. An equivalent version of this help file in HTML help format is also provided: Malibu.chm, for use within the MSVC context.

Innovative Integration Technical Support

Innovative includes a variety of technical support facilities as part of the Malibu toolset. Telephone hotline supported is available via

Hotline (805) 578-4260 8:00AM-5:00 PM PST.

Alternately, you may e-mail your technical questions at any time to:

techsprt@innovative-dsp.com.

Innovative Integration Web Site

Additional information on Innovative Integration hardware and the Malibu Toolset is available via the Innovative Integration website at www.innovative-dsp.com

Typographic Conventions

This manual uses the typefaces described below to indicate special text.

Typeface	Meaning
`Source Listing`	Text in this style represents text as it appears onscreen or in code. It also represents anything you must type.
Boldface	Text in this style is used to strongly emphasize certain words.
Emphasis	Text in this style is used to emphasize certain words, such as new terms.
Cpp Variable	Text in this style represents C++ variables
Cpp Symbol	Text in this style represents C++ identifiers, such as class, function, or type names.
KEYCAPS	Text in this style indicates a key on your keyboard. For example, "Press **ESC** to exit a menu".
Menu Command	Text in this style represents menu commands. For example "Click View | Tools | Customize"

Installation

This chapter describes the software and hardware installation procedure for Windows2000, WindowsXP and Vista. Development under other versions of Windows is not supported at this time, though drivers are provided to allow runtime operation under Windows9x, and WindowsME.

> **Do *NOT*** install the hardware card into your system at this time. This will follow the software installation.

Host Hardware Requirements

The software development tools require an IBM or 100% compatible Pentium IV- class or higher machine for proper operation. An Intel-brand processor CPU is *strongly recommended*, since AMD and other "clone" processors are not compatible with the Intel MMX and SIMD instruction-set extensions which the Armada and Malibu Host libraries utilize extensively to improve processing performance within a number of its components. The host system must have at least 128 Mbytes of memory (256MB recommended), 100 Mbytes available hard disk space, and a DVD-ROM drive. Windows2000 or WindowsXP (referred to herein simply as *Windows*) is required to run the developer's package software, and are the target operating systems for which host software development is supported.

Software Installation

The development package installation program will guide you through the installation process.

> **Note:** Before installing the host development libraries (VCL components or MFC classes), you must have Microsoft MSVC (version 7 or later) and/or Borland C++ Builder (version 6) installed on your system, depending on which of these IDEs you plan to use for Host development. If you are planning on using these environments, it is imperative that they are tested and known-operational before proceeding with the library installation. Additionally, you must install Texas Instruments Code Composer Studio prior to installation of the development package software if you are installing a DSP board (Code Composer Studio installation is not necessary for streaming boards such as ChicoPlus or DAQ). If these items are not installed prior to running the Innovative Integration install, the installation program will not permit installation of the associated development libraries. However, drivers and DLLs may be installed to facilitate field deployment.

You must have **Administrator Privileges** to install and run the software/hardware onto your system, refer to the Windows documentation for details on how to get these privileges.

To begin the installation, start Windows. Shut down all running programs and disable anti-virus software. Insert the installation **DVD**. If Autostart is enabled on your system, the install program will launch. If the DVD does not Autostart, click on Start | Run... Enter the path to the SETUP.EXE program located at the root of your DVD-ROM drive (i.e. E:\SETUP.EXE) and click "OK" to launch the setup program.

After launching Setup.exe, you will be presented with the following screen.

Using this interface, specify which product to install, and where on your system to install it.

1) Select the appropriate product from the Product Menu.

2) Specify the path where the development package files are to be installed. You may type a path or click "**Change**" to browse for, or create, a directory. If left unchanged, the install will use the default location of "C:\Innovative".

3) Typically, most users will perform a "Full Install" by leaving all items in the "**Components to Install**" box checked. If you do not wish to install a particular item, simply uncheck it. The Installer will alert you and automatically uncheck any item that requires a development environment that is not detected on your system.

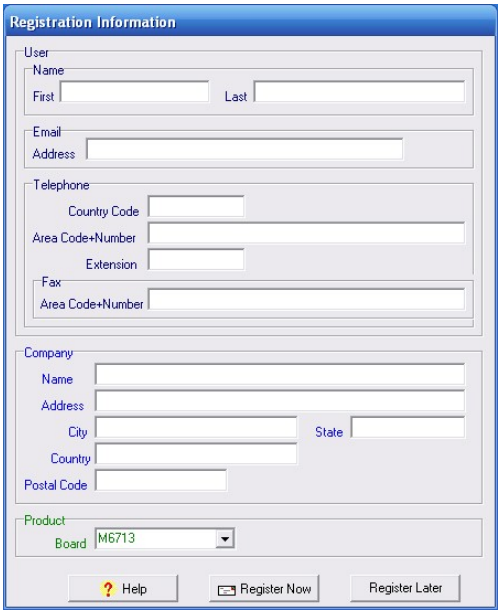
4) Click the Install button to begin the installation.

Note: The default “Product Filter” setting for the installer interface is “Current Only” as indicated by the combo box located at the top right of the screen. If the install that you require does not appear in the “Product Selection Box” (1), Change the “Product Filter” to “Current plus Legacy”.

Each item of the checklist in the screen shown above, has a sub-install associated with it and will open a sub-install screen if checked. For example, the first sub-install for “Quadia - Applets, Examples, Docs, and Pismo libraries” is shown below.

The installation will display a progress window, similar to the one shown below, for each item checked.

Tools Registration



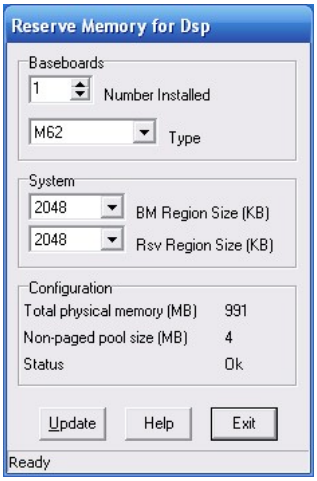
At the end of the installation process you will be prompted to register. If you decide that you would like to register at a later time, click "Register Later".

When you are ready to register, click Start | All Programs | Innovative | <Board Name> | Applets. Open the New User folder and launch **NewUser.exe** to start the registration application. The registration form to the left will be displayed.

Before beginning DSP and Host software development, you must register your installation with Innovative Integration. Technical support will not be provided until registration is successfully completed. Additionally, some development applets will not operate until unlocked with a passcode provided during the registration process.

It is recommend that you completely fill out this form and return it to Innovative Integration, via email or fax. Upon receipt, Innovative Integration will provide access codes to enable technical support and unrestricted access to applets.

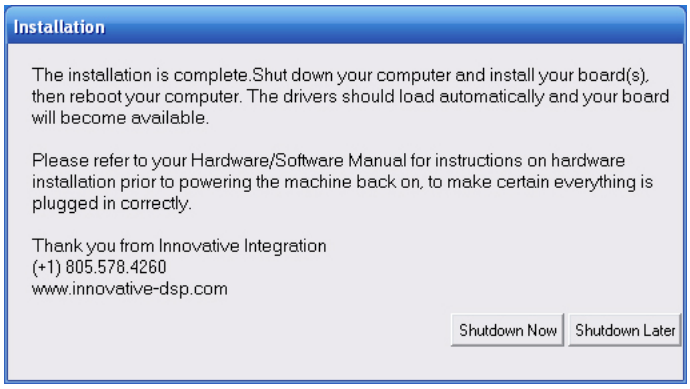
Bus Master Memory Reservation Applet.
At the conclusion of the installation process, **ReserveMem.exe** will run (except for SBC products). This will allow you to set the memory size needed for the busmastering to occur properly. This applet may be run from the start menu later if you need to change the parameters.



For optimum performance each Matador Family Baseboard requires 2 MB of memory to be reserved for its use. To reserve this memory, the registry must be updated using the ReserveMem applet. Simply select the **Number of Baseboards** you have on your system, click **Update** and the applet will update the registry for you. If at any time you change the number of boards in your system, then you must invoke this applet found in Start | All Programs | Innovative | <target board> | Applets | Reserve Memory.

After updating the system exit the applet by clicking the **exit** button to resume the installation process.

At the end of the install process, the following screen will appear.


Installation
The installation is complete.Shut down your computer and install your board(s), then reboot your computer. The drivers should load automatically and your board will become available.
Please refer to your Hardware/Software Manual for instructions on hardware installation prior to powering the machine back on, to make certain everything is plugged in correctly.
Thank you from Innovative Integration
(+1) 805.578.4260
www.innovative-dsp.com
Shutdown Now
Shutdown Later

Click the "Shutdown Now" button to shut down your computer. Once the shutdown process is complete unplug the system power cord from the power outlet and proceed to the next section, "Hardware Installation."

Hardware Installation

Now that the software components of the Development Package have been installed the next step is to configure and install your hardware.

JTAG Emulator Hardware Installation (for DSP boards Only)

First, the emulator hardware must be configured and installed into your PC. The emulator hardware is described in the table below:

Type	Features
Pod-based	Uses a special ribbon cable with integrated line drivers to connect the target DSP emulation signals to the JTAG debugger card. Usable on 3.3 volt or 5 volt designs. (Including 'C54x and 'C6x)

PCI Pod-Based Emulator Installation

To install the PCI pod based emulator, follow the instructions below:

5) Perform the board installation in an “ESD” or static safe workstation employing a static-dissipative bench mat. Wear a properly grounded wrist strap or other personal anti static device. Stand on an anti static mat or a static-dissipative surface.

6) Shut down Windows, power-off the host system and unplug the power cord.

7) Touch the chassis of the host computer system to dissipate any static charge.

8) Remove the card from its protective static-safe shipping container, being careful to handle the card only by the edges.

9) Touch the chassis of the PC to dissipate any built up static charge.

10) Securely install the JTAG board in an available PCI slot in the host computer.

11) Connect the JTAG pod to the host-pod cable. Connect the host-pod cable to the connector located on the end bracket of the JTAG PCI plug-in board.

Baseboard Installation

To install the baseboard:

12) Perform the board installation in an “ESD” or static safe workstation employing a static-dissipative bench mat. Wear a properly grounded wrist strap or other personal anti static device. Stand on an anti static mat or a static-dissipative surface.

13) Shut down Windows and power-off the host system and unplug the power cord

14) Touch the chassis of the host computer system to dissipate any static charge.

15) Remove the card from its protective static-safe shipping container, being careful to handle the card only by the edges.

16) Touch the chassis of the PC to dissipate any built up static charge.

17) Connect the 14-pin connector on the JTAG PCI pod to the DSP board JTAG connector (Non-DSP board users skip this step).

18) Securely install the baseboard into an available PCI slot in the host computer.

IMPORTANT: Many of our high speed cards, especially the PMC and XMC Families, require forced air from a fan on the board for cooling. Operating the board without proper airflow may lead to

improper functioning, poor results, and even permanent physical damage to the board. These boards also have temperature monitoring features to check the operating temperature. The board may also be designed to intentionally fail on over-temperature to avoid permanent damage. See the specific hardware information for airflow requirements.

A Few Considerations BEFORE Power-up

Double-check all connections before applying power. Ensure that the JTAG and baseboard cards seated correctly in the slot.

It cannot be overemphasized:
Double check your cabling **BEFORE** connecting to the baseboard. **DO NOT** hot plug the cables. Hot plugging cables can cause latch-up of components on the card and damage them irreparably. Be aware that the cables to analog inputs are an important part of keeping signals clean and noise-free. Shielded cables and differential inputs (where applicable) help to control and reduce noise.

After Power-up
After completing the installation, boot your system into Windows.

Innovative Integration boards are plug and play compliant, allowing Windows to detect them and auto-configure at start-up. Under rare circumstances, Windows will fail to auto-install the device-drivers for the JTAG and baseboards. If this happens, please refer to the "TroubleShooting" section.

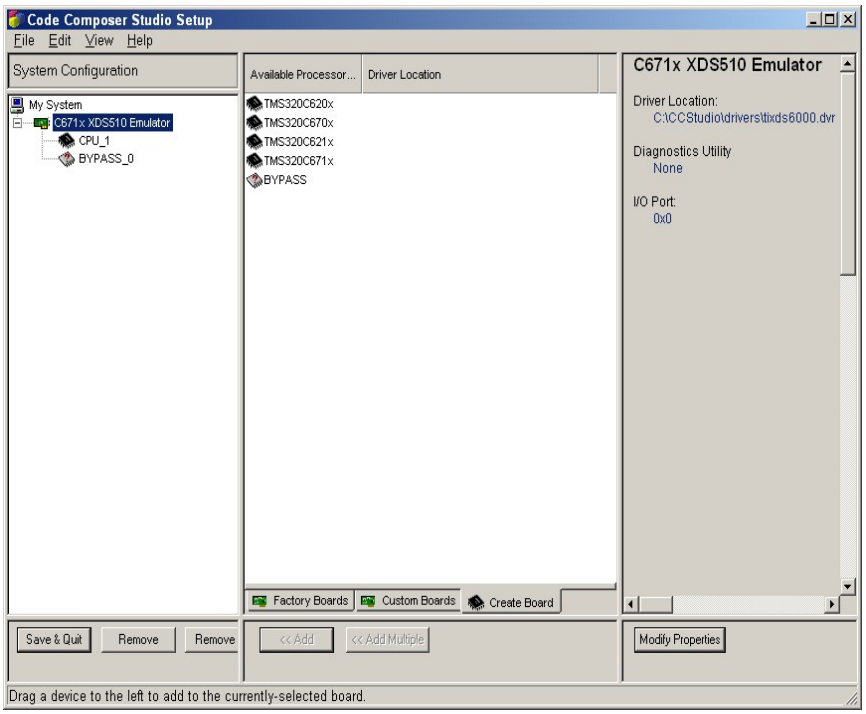
The next section is NOT used with the Non-DSP products. If the board you are installing is a Non-DSP product, the installation is complete.

Code Composer Studio Setup with II Jtag

To setup Code Composer Studio and activate the Innovative Integration-supplied CodeHammer JTAG board driver, the Code Composer Studio Setup Utility must be run. Since the Code Hammer debugger is XDS510- compatible, Code Composer Studio setup must be configured to use the XDS510 driver for the C6xxx. To do this:

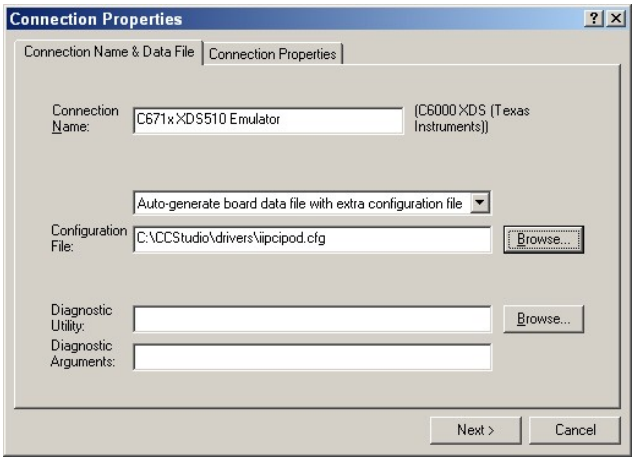
19) Launch the Code Composer Studio Setup Utility and remove the default simulator driver from the System Configuration. (right click the default simulator in the "System Configuration" pane and select "Remove")

20) Click the C6xxx XDS driver from the “Available Emulator Types” control within the setup utility and drag it into the “System Configuration” control.



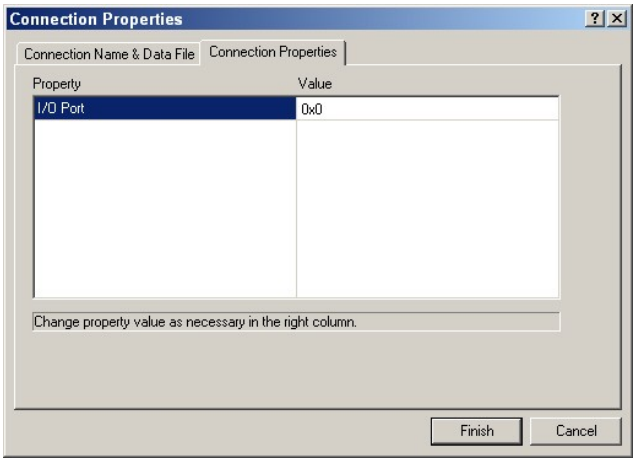
21) Once your emulator is added, a list of Available Processors is presented. Add the appropriate processors for your board as shown in the example. The example shows a set-up that is configured for an SBC6713e baseboard. The C671x emulator is selected as the baseboard uses the TMS320C6713 DSP.

22) Right-click on the C6xxx XDS emulator in the “System Configuration” Pane and select “Properties…” to invoke the **Connection Properties** Dialog for the driver.



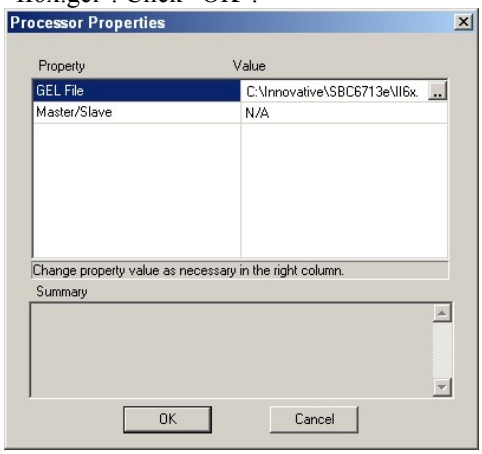
23) Under the “Connection Name & Data File” tab, the “**Connection Name**” edit box should match the emulator selected in the “**System Configuration**” Pane of the previous window. Change the “**Configuration File**” combo box to Auto-generate board data file with extra configuration file. Change the “**Configuration File**” edit box to "<drive>:\Cstudio\Drivers\IIPciPod.cfg". *<drive> is the letter for the drive onto which CCS is installed.*

24) Click the "**Connection Properties**" tab. Set the I/O port value for the driver to virtual device address "0x0" and click "**Finish**".



25) The main Code Composer Studio Setup window is now back in focus. The processor must now be configured. To do this: select the processor as shown in the "**System Configuration**" Pane (in our example "CPU_1"). Right click "CPU_1" and select "Properties…".

26) The "**Processor Properties**" screen will be presented. Click GEL File, click the ellipsis (…) and navigate to the Innovative Integration board install directory (typically C:\Innovative\BoardName) and select "II6x.gel". Click "OK".

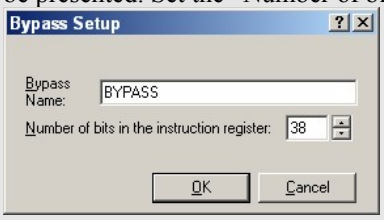


27) Click "**Save & Quit**" to save the configuration and exit the setup tool. You will then be prompted to launch **Code Composer Studio**.

Note: For multi-target boards such as the Quadia, one processor should be added for each device in the JTAG scan path.

Note: The SBC6713e has (2) DSPs, a C6713 and a DM642. Typically the DM642 should be set to "BYPASS" by selecting "BYPASS" from the "Available Emulator Types" control within the setup utility and drag it into the "System Configuration" control. Once this is done, the following screen will

be presented. Set the “Number of bits in the instruction register” to “38” and click “OK”



If you encounter difficulty launching CCS

28) Run the **JtagDiag.exe** utility (Start | All Programs | Innovative | Common Applets | JTAG Diagnostics) to reset the debugger interface.

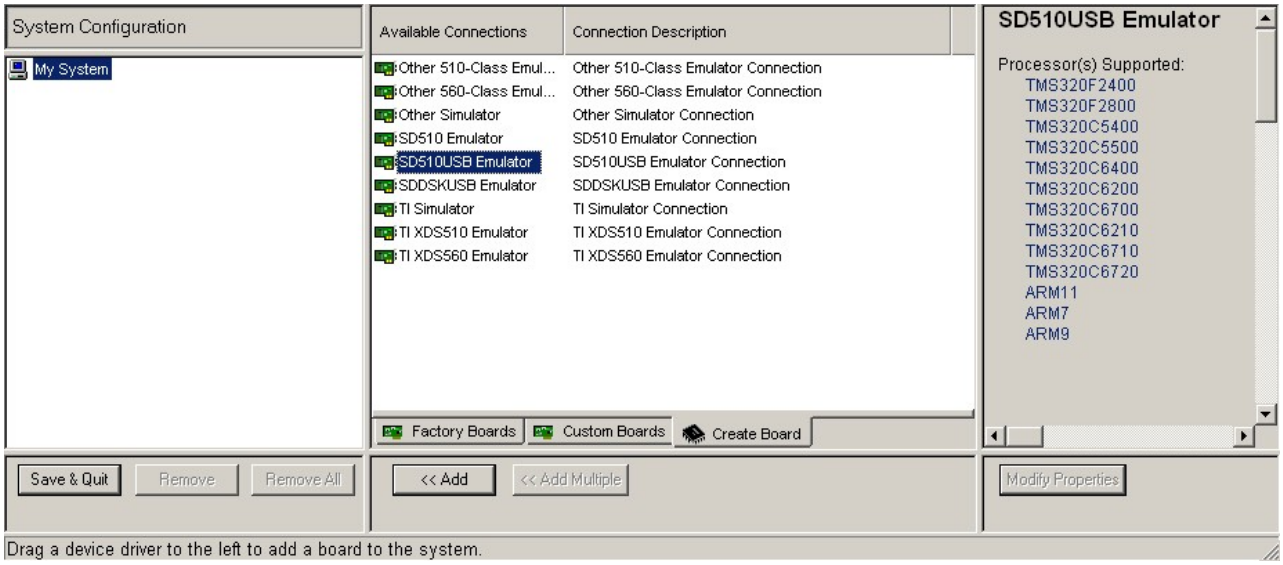
29) Run the board **Downloader utility** (Start | All Programs | Innovative | <Board Name> |<Applets> — Open the **Downloader Folder** and double click “**Downloader.exe**”) and press the Boot button (*Light Bulb icon*), to boot a default target application.

30) Restart Code Composer Studio.

Setting up for a single processor with Spectrum Digital USB Jtag.

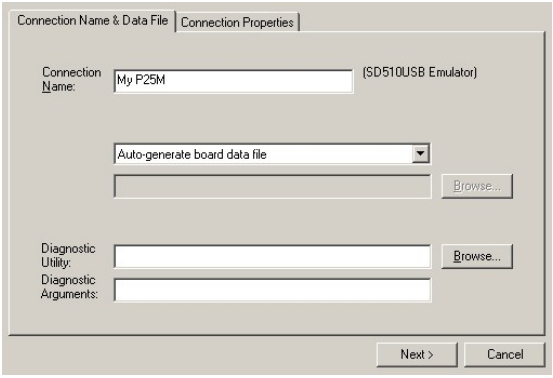
First remove any previous setups in the CCS Setup application.

Add one of the USB SD type driver.



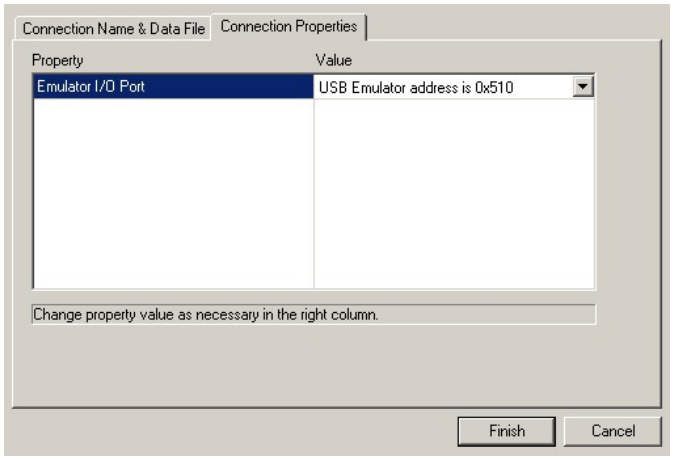
You will see the following screen.

Fill out the name of the board you are using, this can be any name you like.



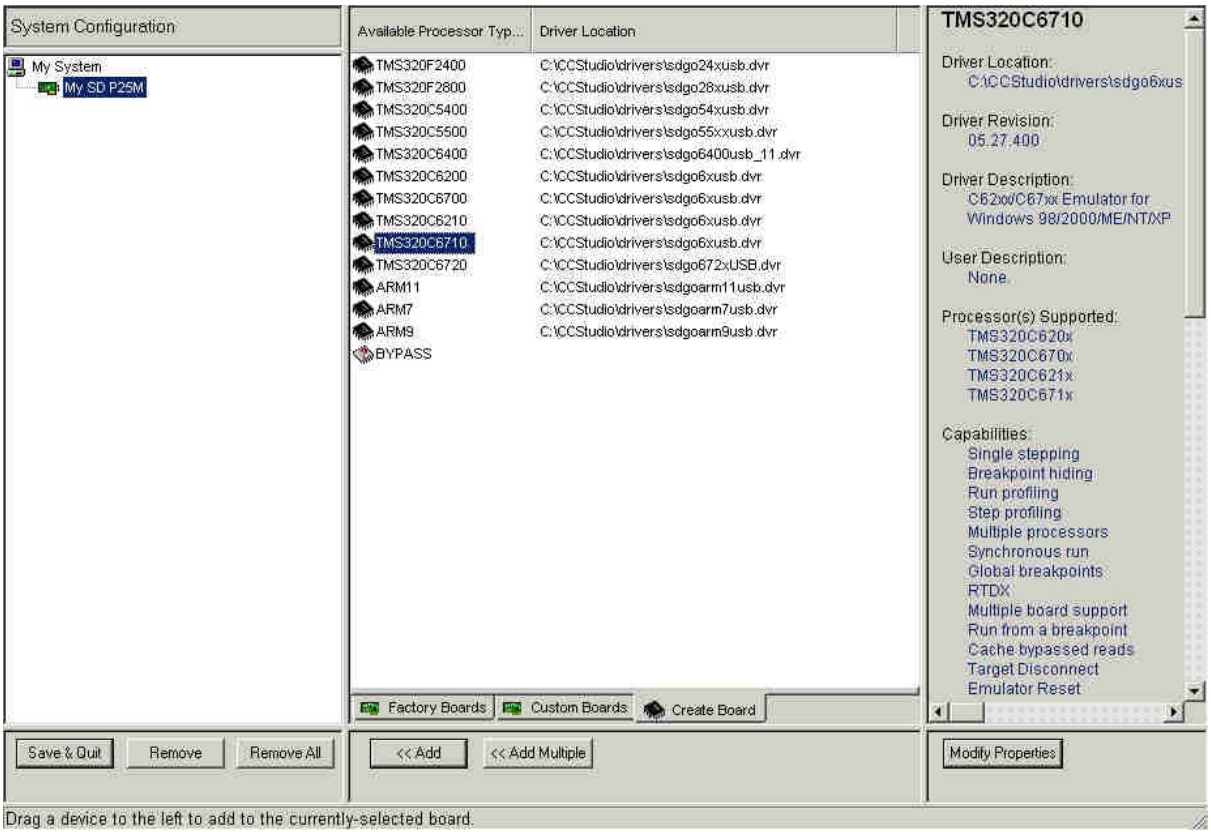
Hit next or move to the next tab

This address should match up with the address in the SdConfig.exe utility

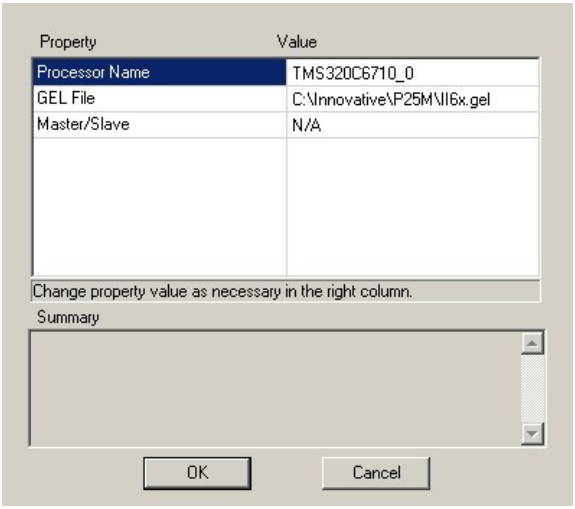


Now we add a processor

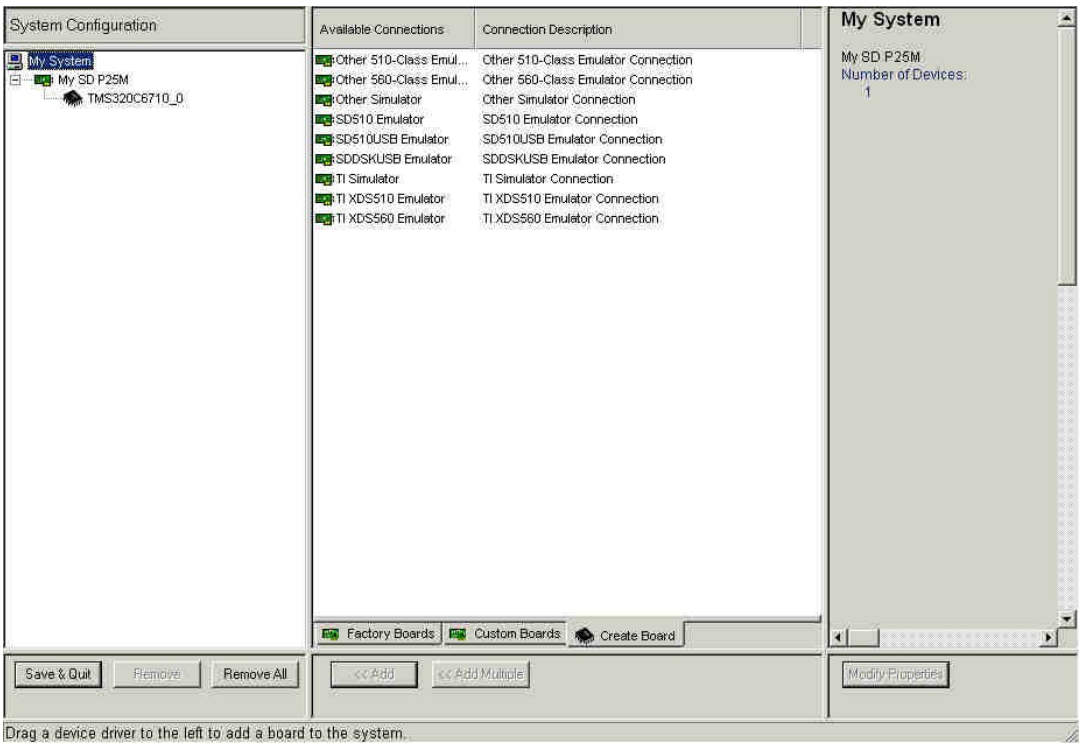
Each if the II boards have different processors so match up the closest one for your board.



Use the property sheet to find the Gel file from Innovative for your specific board.



Your system will look similar to this. Save the configuration and quit.



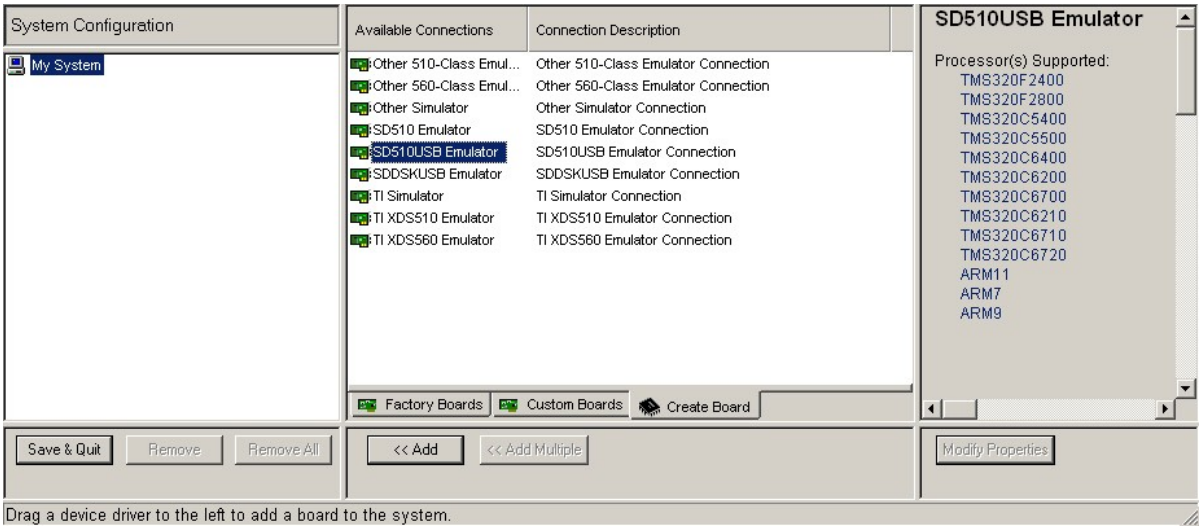
Setting up for Multi Processors with Spectrum Digital USB Jtag.

For the multi-processor setups use the following type setup. This includes the SBC6713e, Quadia, Q6x type Innovative boards.

The SBC6713e board shown will be similar in setup with the other boards. The differences will be in the types of processors and the number added.

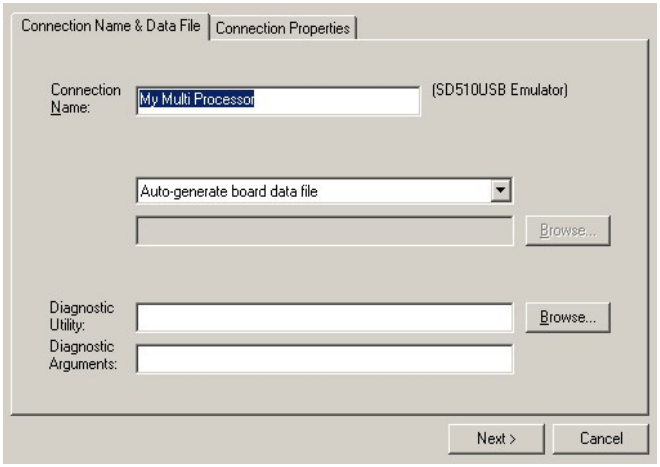
First remove any previous setups in the CCS Setup application.

Add one of the USB SD type driver.



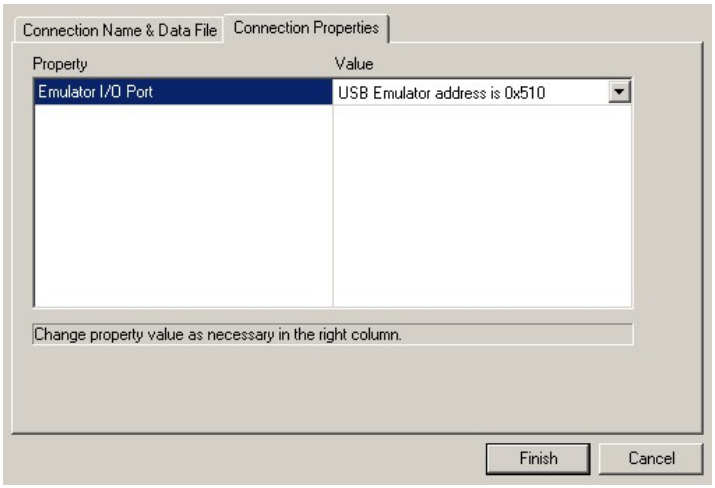
You will see the following screen.

Fill out the name of the board you are using, this can be any name you like.



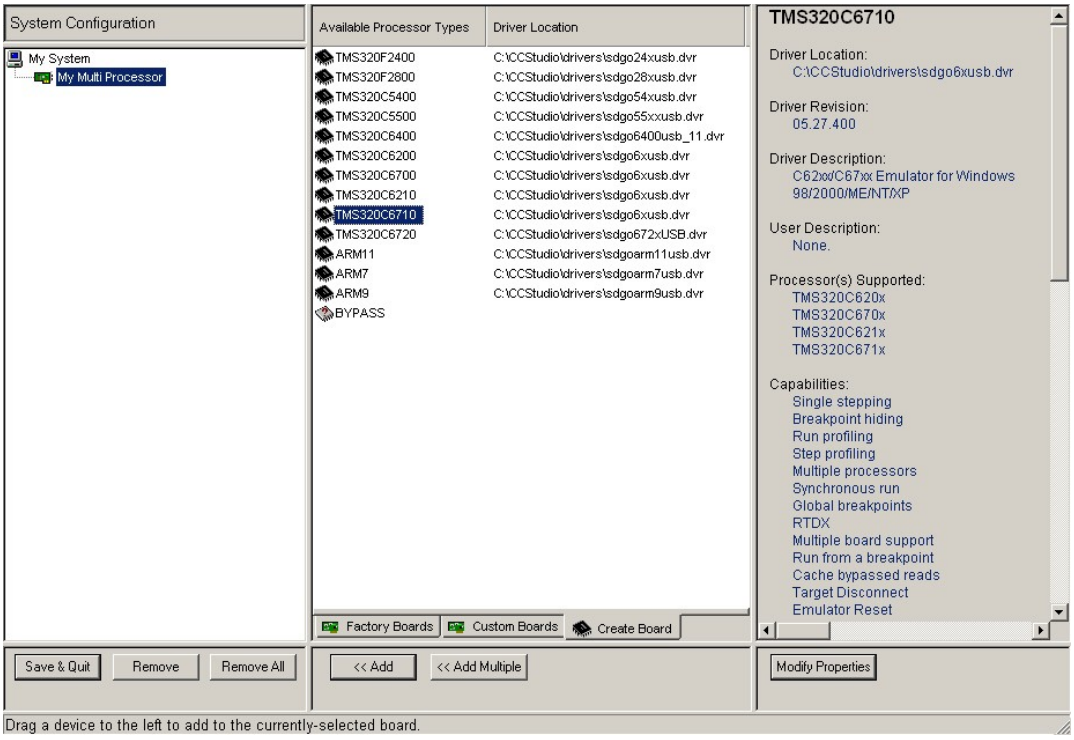
Hit next or move to the next tab

This address should match up with the address in the SdConfig.exe utility

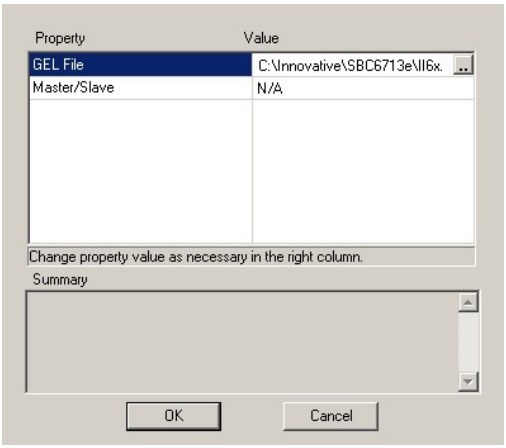


Now we add a processor

Each if the II boards have different processors so match up the closest one for your board.

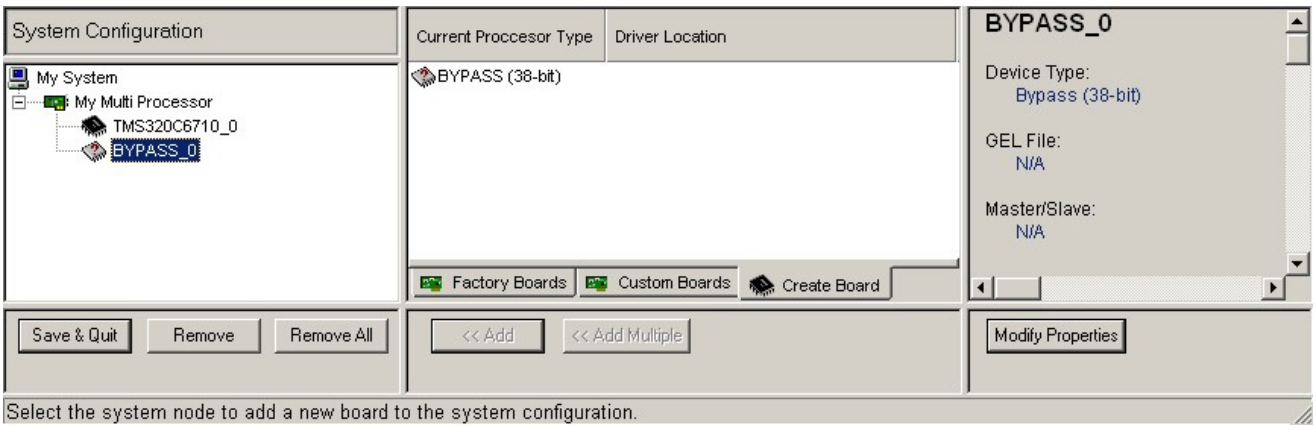


Use the property sheet to find the Gel file from Innovative for your specific board.



Close the processor and choose another processor. This will be a bypass for the DM642. Set the bypass for 38 bits. (For TMS6713 bypass use 42 bits on the first processor, the second processor will be a 64xx and the gel file from II for the DM642). For the Quadia use another C6400 type processor totaling 4 processors. All 4 will use the same GEL file from II.

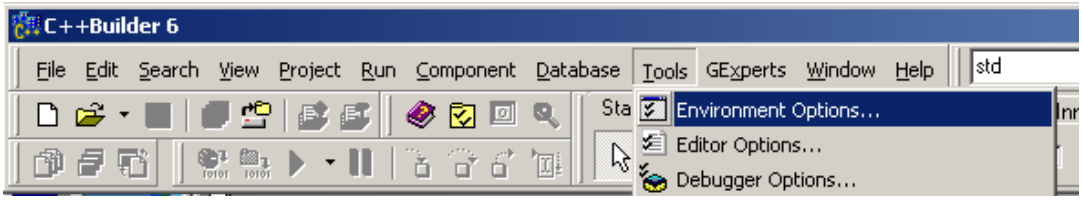
Your system will look similar to this. Save the configuration and quit.



Borland Builder Setup and Use

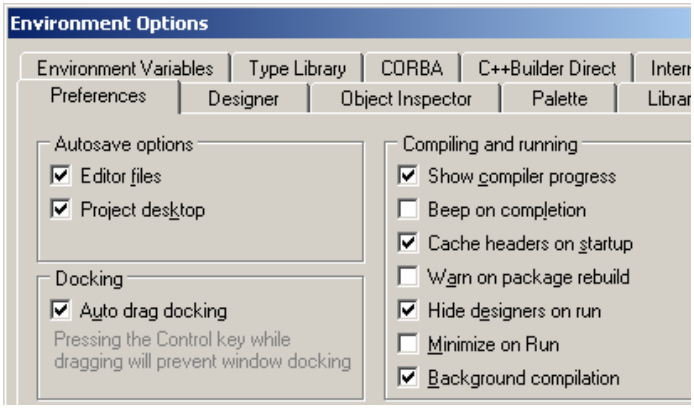
Following the normal installation of the Innovative Integration toolset components, numerous VCL components and C++ classes are automatically added to the BCB IDE. Additionally, Innovative recommends that the following IDE and project options be manually changed in order to insure simplified use and proper operation:

Automatic saving of project files and forms during debugging



31) Select Tools | Environment Options... from the main BCB toolbar.

32) This will invoke the **Environment Options** dialog:



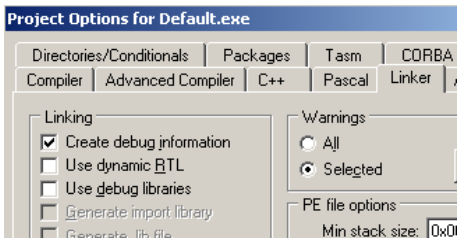
33) Click the Preferences Tab

34) Check "**Editor files**" and "**Project desktop**" under "Autosave Options" so that project files are automatically saved each time a project is rebuilt and debugged.

35) Click "OK"

Static-binding of built executables.

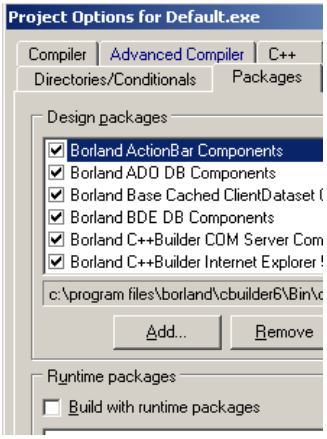
36) Click on Project | Options on the main BCB toolbar to invoke the **Project Options dialog**.



37) Click the **Linker** tab.

38) Uncheck the "Use Dynamic RTL" checkbox.

39) Next, click on the Packages tab and uncheck the "Build with runtime packages" checkbox.



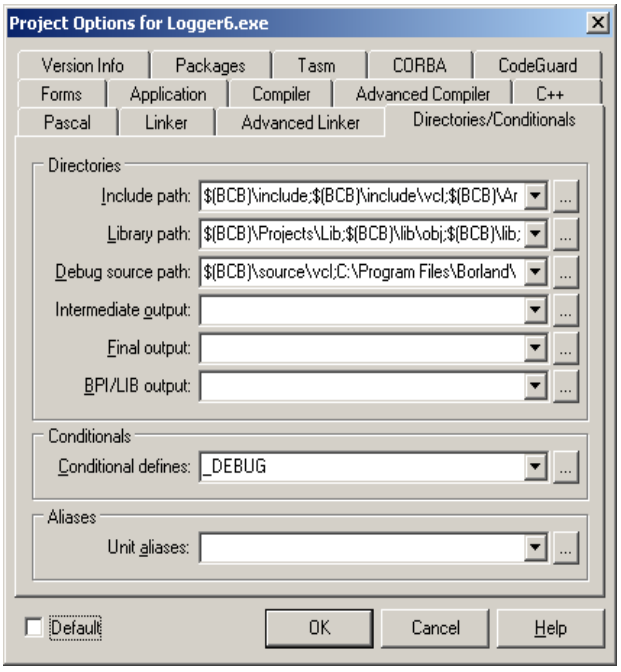
These options insure that projects are built with minimal dependencies on external DLLs. See the FAQ "**What DLLs do I have to deploy with my newly created executable**" in the Troubleshooting chapter for details on which DLLs must be deployed with user-written executables.

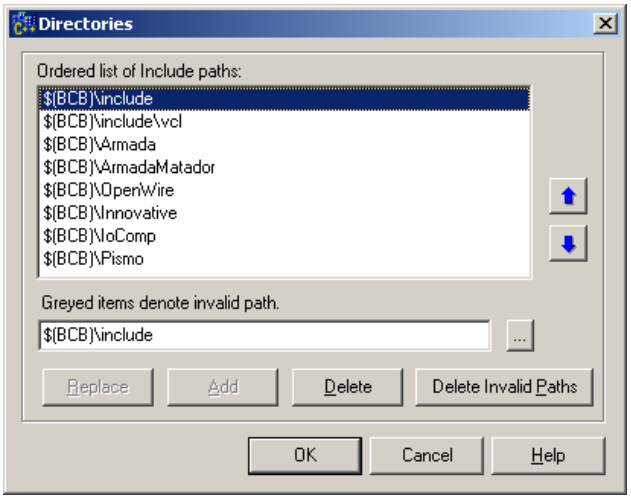
Appropriate library and include paths.

40) Click on the "Directories/Conditionals" tab.

41) Click the ellipses next to the Include Path edit box to invoke the Include Path editor dialog. Add entries for Armada, ArmadaMatador, OpenWire, IoComp and Pismo, as shown below, then click **OK** to accept these edits.

42) Next, click on the ellipses next to the Library Path edit box to invoke the Library Path editor dialog. Add entries for Armada, ArmadaMatador, OpenWire, IoComp and Pismo, as shown below, then click **OK** to accept these edits.





These changes insure that the standard Armada headers and object files are available to projects during compilation. Note that these paths may either be added to the default BCB project, by editing these options without first opening a specific project, or to specific projects after opening them. The advantage of the former is that the settings are automatically present on all subsequently-created projects.

Quixote Baseboard Overview

Quixote Hardware Overview

All Velocia baseboards feature the Texas Instruments TMS320C6416 digital signal processor, Xilinx FPGAs and extensive peripheral feature set to support demanding signal processing applications. The tight coupling of the DSP, FPGA and peripherals make these boards well suited for a variety of applications such as communications baseband processing, ultrasound, RADAR and many data acquisition applications.

The following table gives an overview of the Quixote baseboard.

Table 1. Quixote Feature Matrix

Velocia Family Card	DSP	IO Features	Example Applications
Quixote	TI TMS320C6416	2 channels of 14-bit 105 MSPS A/D 2 channels of 14-bit 105 MSPS DAC PMC site	Wireless test equipment, RADAR, signal intelligence

The Velocia family of DSP cards have similar, though not identical features. Since the cards are all built around the ‘6416 DSP and Xilinx FPGAs, many parts of the card architecture are similar as is the software development kit.

Quixote Block Diagram

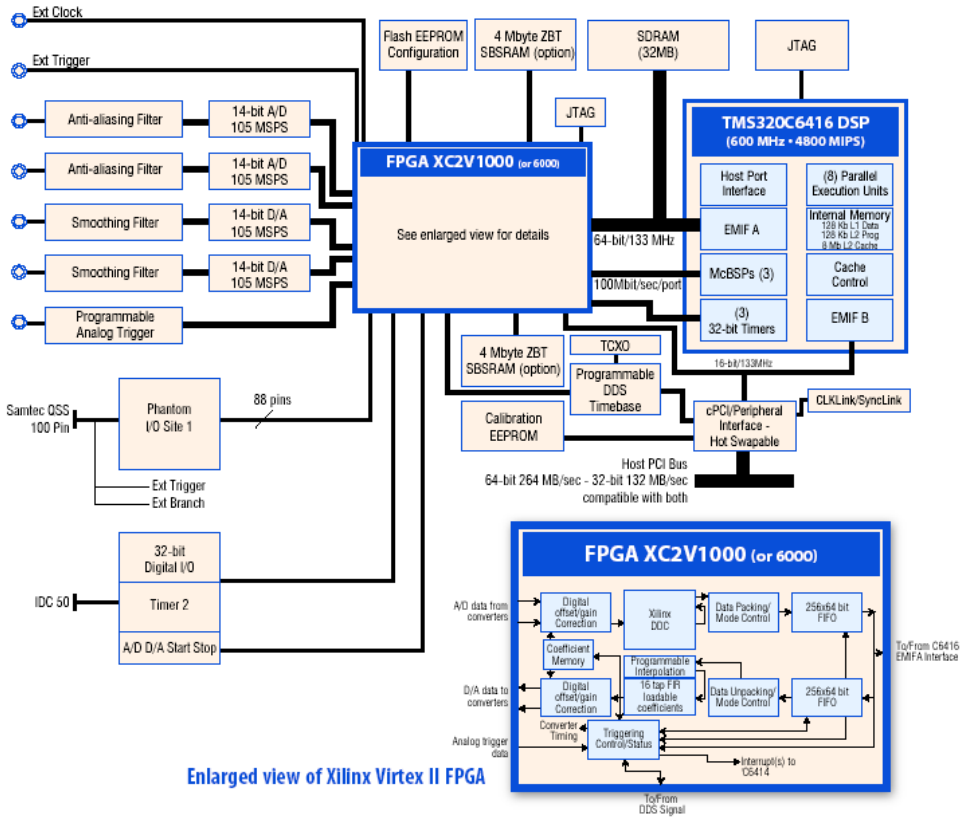


Figure 1. Block Diagram for the Quixote Baseboard

Baseboard Enumeration

Velocia baseboards are Plug-and-Play PCI devices which require a device driver for Win2K and WinXP. The same device driver, *iixwdm.sys*, is used for all baseboards. The appropriate driver is installed as part of the normal installation process for each baseboard. The driver accepts the resource assignments given for the board and configures the software to use them, making the board fully recognized by Windows.

On the Quixote, the C6416 CPU is also present on the PCI bus, and enumerates as a device of its own. In addition, a device in the PMC slot will also enumerate as a separate device.

These devices reserve a block of contiguous, physical memory for use as a region for bus-master transfers. This block ranges from 2-8 MB in size. Each baseboard will reserve its own region that will not be available for Windows applications or other boards.

Under Win2k and WinXP, reserving this space may require the raising of the reserved system memory ceiling whenever:

1. One or more baseboards are installed into the PC system (including initial installation!)

2. Operating at high data acquisition rates or with large channel-count systems

ReserveMemDsp.exe, is provided to support manual adjustment of these registry properties. See the Applets chapter for details.

Multiple Baseboards

When more than one baseboard is installed in a system there needs to be a means of uniquely addressing each installed baseboard. Each board is assigned an integer identifier, the *target number,* which corresponds to the baseboard's location on the PCI bus, so the assignment for a particular arrangement of boards is fixed regardless of changes to the rest of the system. If additional boards are added, target assignments for all boards may change.

Unfortunately, the relationship between PCI slots and driver-assigned target numbers is system-dependent. So, a means of associating a target number with a baseboard installed in a particular PCI slot is needed. To determine the slot-to-target-number associations, the baseboard has an LED that can be illuminated by a software access from the host.

PCI Interrupt Configuration and Compatibility

Each C64x DSP residing on the PCI bus requires from at least 2 MBytes of Host PC memory space and one interrupt. Each C64xDsp instance maps two memory spaces into Host memory – one corresponding to target SDRAM memory and another corresponding to target peripherals. Additionally, a single PCI interrupt is consumed by each DSP.

These resources are requested as part of the Plug-n-Play boot operation and are not programmable. Failure to allocate these resources will cause erratic performance. Windows will NOT report the failure to allocate resources and the user should use care during installation that the resources are available and properly allocated. The proper allocation of resources to the card may be checked on the system properties page under My Computer | Control Panel | Properties. Each instantiated C64x DSP should report that there are no conflicts.

Velocia baseboards are compatible with PCI specification revision 2.1 and have been tested with a variety of systems for compatibility. The Velocia device driver shares interrupts properly as required by the PCI specification..

In use, the Host-Side Malibu libraries handle all the details of interrupt configuration and response. The host application receives notification when data is available or required for data streaming and messaging purposes.

Quixote Logics

There are two programmable logic devices on the Quixote baseboard. The FPGA logic controls the board and the analog logic. This logic must be loaded at least once per Windows session. The second logic is an FPA that controls the PCI interface. This is reloadable via software, but should not have to be loaded except for when logic updates occur.

Quixote Host Software Overview

The Malibu library provides C++ Classes to operate the Quixote baseboard. The object manages the allocation of resources from Windows for the board. All operations on the Quixote and the TI C64x CPU are performed through these 'baseboard' objects, each controlling one of the boards on the PCI bus.

The Innovative::Quixote object

The Quixote object controls the functions of the baseboard exclusive of the DSP. In Malibu, this object performs a supporting role, as I/O passes through the DSP.

Initialization

There are several methods involved in setting up the baseboard. The most important is the Open() method. This is the time when the object attaches to the hardware and allocates its resources. The Target ID should have been set before this method is called.

Table 2. Quixote Initialization Methods

Method	Functionality
Open()	Attach to hardware device.
Close()	Detach from hardware device.
Led()	Blink board ID LED device.
Target()	Set Target Number.
Reset()	Reset the hardware.

Logic Configuration

The Quixote baseboard has a configurable FPGA that must be loaded once per session. The Logic() method returns a Malibu Interface, *Innovative::IUsesVirtexFpgaLoader,* which is used to load the logic part. The ConfigureFpga() method takes the name of the file to read the logic image from. This file must be in EXO data format.

The loader interface has a number of event callbacks that can be used to display feedback on the progress of the load. See the Malibu library documentation on the use of these events.

Table 3. *IUsesVirtexFpgaLoader* Configuration Methods

Method	Description
IsConfigured()	Returns true if the logic device has been successfully configured, false otherwise.
ConfigureFpga()	Resets the logic device, then parses and downloads the specified EXO image into the baseboard logic device.

The Quixote installation provides an applet that loads the FPGA logic.

FPA Configuration
The FPA logic can also be updated via software. The VsProm() method returns an *Innovative::FpaLoadThread* interface that can be used to update the FPA.

The Quixote installation provides an applet that loads the FPA logic.

Target to Host Communication
The streaming bus master link in the Quixote baseboard is not used under Malibu and is disabled. The mailbox link is also not used.

The Innovative::C64xDsp object

The C64xDsp object manages the access to a C6416 DSP on the PCI bus. It allows COFF files to be loaded and executed on the target and manages the communication links to the target after execution has begun.

The C64xDsp class represents the C64x processor on any baseboard. The same class is used for the processors on the Quadia baseboard as well as the Quixote.

Initialization
There are several methods involved in setting up the DSP object. The most important is the Open() method. This is the time when the object attaches to the hardware and allocates its resources. The Target ID should have been set before this method is called.

Table 4. C64xDsp Initialization Methods

Method	Functionality
Open()	Attach to hardware device.
Close()	Detach from hardware device.
Target()	Set Target Number.
Reset()	Reset the hardware.

COFF File Downloading
At power-up, the Velocia baseboard DSP has no program of any kind running on it. In order to have the hardware perform any action, a software program must be downloaded from the host and run.

The C64xDsp class supports downloading of COFF Files to the target and executing target programs. The Cpu() method returns a *Innovative::IUsesOnboardCpu* interface class that enables COFF loading. Calling the DownloadCoff() method with the name of a COFF file begins downloading the file. When the load is complete the target processor is started and the program begins execution.

The Interface also includes events for monitoring the progress of the download. See the Malibu library documentation for more details on this interface.

Table 5. *IUsesOnboardCpu* Loading Methods

Method	Description
Boot()	Resets the DSP and all DSP-addressable peripherals, but not the Host interface. Ensures that the DSP is in a state suitable for subsequent JTAG emulation.
DownloadCoff()	Resets the DSP and downloads specified COFF executable, then launches it

Due to restrictions in the TI C64x DSP architecture, it is not possible to successfully connect the JTAG emulator to a DSP target running a DSP/BIOS application program. Use *Boot()* to place a processor into a benign state, suitable for subsequent JTAG emulation. Then, Connect the processor within Code Composer Studio.

To load a program onto the target, call the *DownloadCoff()* method passing the name of the TI COFF executable file (.out) to download. At the conclusion of the download, the target application begins execution. Be sure to start Code Composer Studio prior to downloading DSP/BIOS-based applications using this method, to avoid the problem above.

Target to Host Communication

The PCI Interface of the C64x DSP allows busmastering of data between the host and target memory. This interface is used to implement a bidirectional interrupt driven link that can efficiently send small command messages and large data packets over the same hardware linkage.

Communication links in Malibu are supported by a data stream class object that links to a hardware object supporting a style of I/O. The C64xDsp class supports the *Innovative::TiBusmasterStream* link. This data stream class is more fully described in the chapter on Target to Host communication.

To use the stream with the C64xDsp, create a stream object and then use the ConnectTo() method of the stream class to associate the stream with the board. From then on, the methods of the stream will use the DSP to send and receive message packets and data packets from the hardware.

Quixote Target Software Overview

The Quixote C64x target uses the Pismo class library. Special features for the Quixote baseboard are described here.

Communication to the Host

There are two modes of communication with the host: a small 16 words message packet and a much larger data packet that uses PCI Busmastering to transfer the data to the host. To send a message packet, create an *IIMessage* object and fill it in with the data you wish to send. Then create a *MessageTransfer* object and call the Send() method with the packet you wish to send. The *MessageTransfer* object's Recv() method waits for a message to arrive, then fills in the IIMessage argument with the data. Since this call waits until a packet arrives, it is best to dedicate a thread for message input and processing.

Packet transfer is similar. Any class derived from *Buffer* can be sent to the host by a *PciTransfer* object's Send() method. The Recv() method blocks until a data packet arrives. As with commands, it is best to dedicate a thread for taking in data packets arriving from the host.

Analog Input Stream Devices
The *AinStream* class is used to get input from the ADCs on the Quixote. To use it, set the buffer size using the Events() method, configure the analog using the AioCtl device accessed by the Device() method, then call Open() to start the streaming. After this, calls to Get() will return when a new block is available. Access the latest block with the Buffer() method. Repeat until you wish to stop data taking, when you call the Close() method to stop data acquisition.

Analog Output Stream Devices
The *DacStream* class is used to output buffers to the DACs. You configure the device and set the buffer size just as in the input stream. After Open() is called , use Put() to insert buffers into the output stream. After a few buffers are loaded for prefilling, the output will begin. The Close() method stops outputting data.

Timebase Objects
A major portion of the configuration of analog I/O is setting the clock and configuring the triggering modes – thus deciding how data taking starts, stops, and when samples are taken. The Quixote encapsulates this configuration in a family of Timerbase objects.

Each timebase is an abstraction of a typical mode of data streaming collected together into an easy-to-operate object. Some typical modes of operation, and their descriptions are tabularized below:

Table 6. Data Acquisition Modes for the Quixote

Operational mode	Description
Basic	Continuously acquire or generate streams. Acquisition or signal playback commences (is triggered) via the start trigger (by default, software command) and terminates via the stop trigger (default, software command).
Framed	Stream a specified number of samples. Streaming commences via the start trigger (by default, a software command) by and terminates upon having streamed a specified number of samples.
Timed	Stream for a specified time interval. Streaming commences via the start trigger (by default, a software command) and terminates upon having streamed for a specified number of microseconds.
Threshold	Stream whenever a specified threshold conditioned is detected. Streaming commences upon detection of a voltage higher than the threshold value on a specified channel, and terminates whenever the voltage dips below the threshold value.
Slave	Streaming is initiated upon receipt of a start trigger on the SyncLink connector. Streaming is terminated upon receipt of a stop trigger on the Synclink connector. One of the three SyncLink clock pins acts as the conversion clock.
Custom	Streaming is initiated via software command, or detection of a threshold condition. Streaming is terminated via software command, reaching a specified frame count, elapsed time limit or detection of a specified threshold condition. The type, polarity and sense of the start and stop triggers, conversion clock source, frame count, threshold criteria and elapsed time limits are all fully user programmable.

All of the timebases above are used in systems consisting of a one or more DSP baseboards. In multi-target systems, one target application acts as the source for trigger and clock information for all boards in the system. In a system consisting of just a single DSP baseboard, target application programs use any timebase other than a Slave timebase to provide clocking information to its onboard peripheral devices, but this timebases trigger and clock will not be routed externally.

Slave timebases are used exclusively within systems consisting of more than one DSP baseboard. Target applications which are to be slaved to a master within a system each employ a SlaveTmb timebase object which configures its streaming peripherals to receive clock and trigger information from one master baseboard within the system via the SyncLink connector.

Timebase Software Objects

All timebase components derive from the base class BasicTmb. This class provides fundamental services, common to all timebase objects. These services include the ability to:

- Query and change the sample rate at which the timebase will generate conversion pulses
- Initiate a start and stop trigger via software
- Automatically trip the start and stop trigger as the Pismo device driver initiates or terminates streaming

Derived timebase objects are tabularized below, listed with their default start and stop trigger and conversion clock sources.

Table 7. Quixote Timebase Objects

Timebase Object	Default Start Trigger	Default Stop Trigger	Default Conversion Clock	Device
BasicTmb	Software	Software	DDS	A/D or D/A
AdcFramedTmb	Software	A/D Frame Count	DDS	A/D
DacFramedTmb	Software	D/A Frame Count	DDS	D/A
AdcElapsedTmb	Software	A/D Frame Timer	DDS	A/D
DacElapsedTmb	Software	D/A Frame Timer	DDS	D/A
AdcThreshTmb	Voltage on specific A/D channel greater than threshold	Voltage on specific A/D channel less than threshold	DDS	A/D
DacDelayTmb	Software	Software	DDS	D/A
MultiTmb	User-specified	User-specified	User-specified	A/D or D/A
SlaveTmb	SyncLink 2 or 4	SyncLink 1 or 3	SyncLink clock channel	A/D or D/A

To use a timebase, instantiate the timebase which most closely matches your streaming requirements. Then optionally customize to suit. Finally, attach the timebase to the device driver and begin streaming. For example, the code fragment below illustrates configuration for continuous acquisition from analog input.

```
// Instantiate A/D driver object
AdcStream Ain;

// Attach timebase to A/D driver
BasicTmb Timebase;
Ain.Device().Attach(Timebase);
```

```
// Enable specified analog pairs
for (int i = 0; i < NumChannels; ++i)
    Ain.Device().Channels().Enabled(i, Status.ChannelMask&(1<<i));

// Size the stream buffers to hold specified number of events
Ain.Events(NumEvents);

// Open the analog stream objects
Ain.Open();
```

To customize the above code to use an external clock source, instead of the default baseboard DDS, simply manipulate the Timebase object dynamically after creation:

```
// Attach timebase to A/D driver
ContinuousTmb Timebase;
Timebase.ClockSource(csExternal);
Ain.Device().Attach(Timebase);

...

// Open the analog stream objects
Ain.Open();
```

The MultiTmb object provides a generic interface to all of the timebase circuitry on the baseboard. All of the features and properties of the baseboard timers are exposed, providing a complex, albeit unrestricted view of the capabilities of the baseboard timebase facilities. In general, it is easier to use one of the focused timebase components, such as ContinuousTmb or FramedTmb as the trigger source within application programs. However, in some rare instances, an unusual combination of features of the onboard timebase circuitry is not exposed or accessible using the simplified timebase components. MultiTmb is provided to address those esoteric needs. While this timebase is the most flexible, it is also the most difficult to configure.

Digital Io

The Quixote digital I/O is controlled by the *UserData* class. It allows the configuration of the lines to input or output and the reading or writing of the data.

Quixote Architecture

Quixote combines a powerful DSP and FPGA processing core with high speed analog IO and other peripherals ideal for software digital radio applications. The following block diagram shows the Quixote architecture and features.

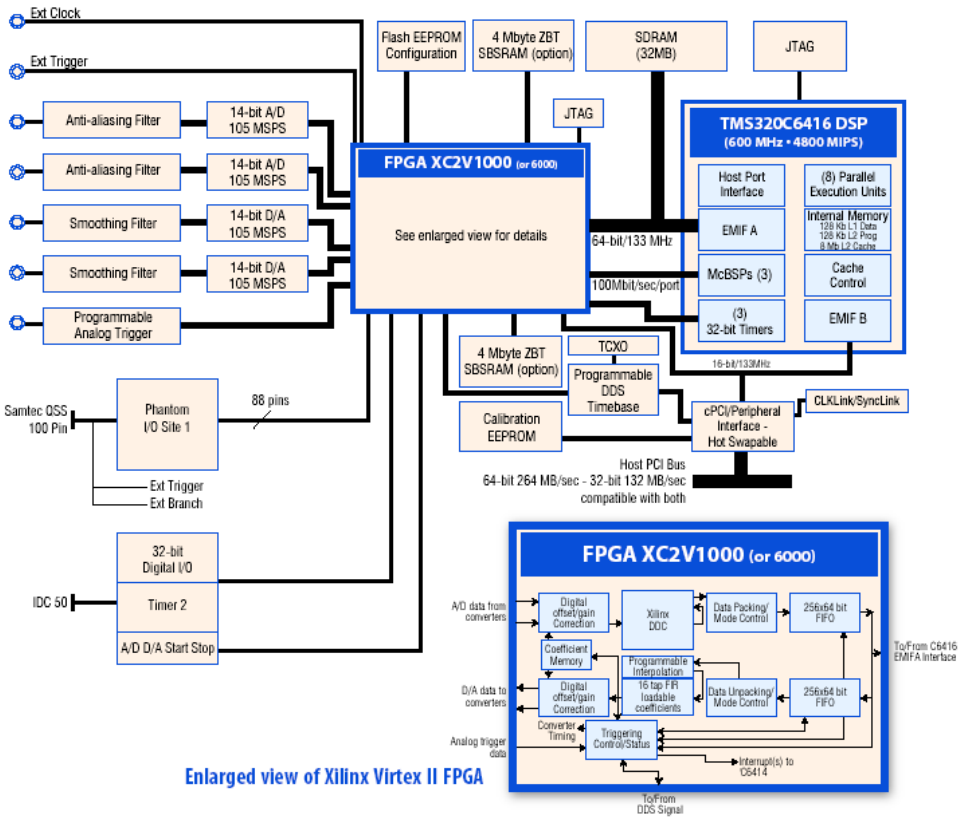


Figure 2. Quixote Block Diagram

Quixote is built around a Texas Instruments TMS320C6416 digital signal processor (DSP) chip coupled with a Xilinx Virtex2 FPGA as the computational engines. The DSP is capable of up to 8000 MIPs processing (1 GHz clock rate) and has dual memory buses, a PCI interface and many on-chip peripherals making it the most powerful programmable DSP available.

The DSPs is complemented with a Xilinx FPGA that is a Virtex2 device. The Virtex2 has many features for both signal processing and high speed computing such as a embedded multipliers, embedded memory, and a logic fabric of 2 M gates (XC2V2000) or 6 M gates (XC2V6000) .

Quixote features a flexible PCI and data architecture suited for high rate, real-time signal processing. The data plane and PCI busses are extensible for system integration and extension to multiple card systems.

Peripherals on the Quixote card include dual 105 MSPS A/D channels, dual 105 MHz D/A channels, a PMC site, timing controls for synchronization and sample rate generation, and system connectivity.

Processing Core

The Quixote processing core is composed of a ‘6416 DSP and a Virtex2 FPGA.

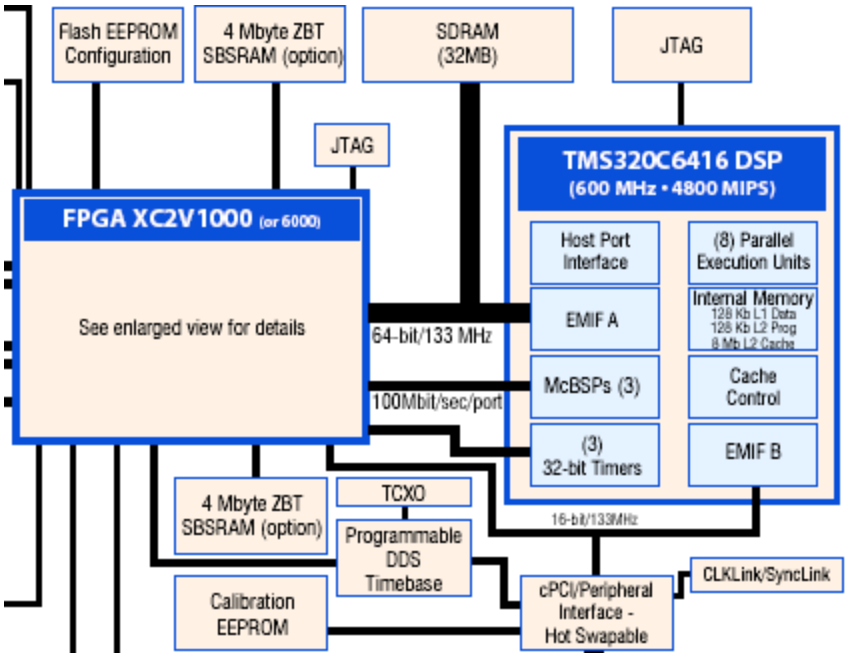


Figure 3. Quixote Computing Core

As can be seen from the figure, the DSP has its private memory on EMIF A and is connected to the FPGA by both EMIF A and EMIF B. The DSP memory is 64 MB of SDRAM running at 100 MHz. The FPGA connection over EMIF A is 64-bit at 100 MHz, providing a maximum rate of 800 MB per second. The connection to the FPGA over EMIF B is 16-bit at 100 MHz, yielding a maximum rate of 266 M bytes per second.

The Virtex2 FPGA provides a powerful computing element in cluster with its dense logic array, embedded memory and DSP features. The FPGA is tightly coupled to the DSPs and PMC. Private J4 connection to the PMC allows modules supporting unique J4 data interface to implement high performance control and data interfaces to the PMC. The FPGA also has private memory comprised of dual 4 MByte synchronous ZBT SRAM memories for use by the FPGA as storage or computational memory.

Connectivity outside of the computing core consists of the PCI bus, the data plane connections from the FPGA and PMC and the digital IO for off-card connections for system expansion and IO communications.

Analog I/O

Quixote has two identical analog IO channels of 105 MSPS analog-to-digital (A/D) converters and two identical channels of 105 MSPS digital-to-analog (D/A) converters. Each channel has a 32K sample FIFO buffer connecting it to the FPGA. Analog channels have filters on the IO for noise reduction.

A variety of clock and trigger sources are supported for analog signal acquisition and generation. On-card timing signal generation is provided using a precision, low-jitter programmable PLL. Triggering modes include external triggers, timed and sample length modes, analog input threshold triggering and DSP driven acquisitions to name a few.

Simplified views of the channels are shown below.

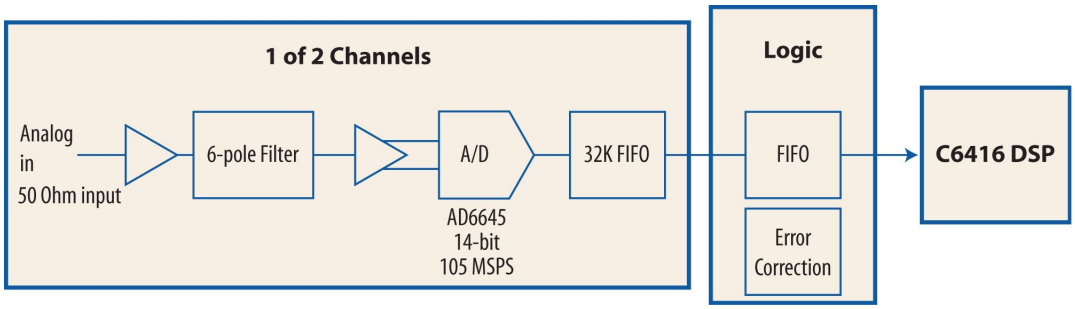


Figure 4. Simplified Quixote A/D channel

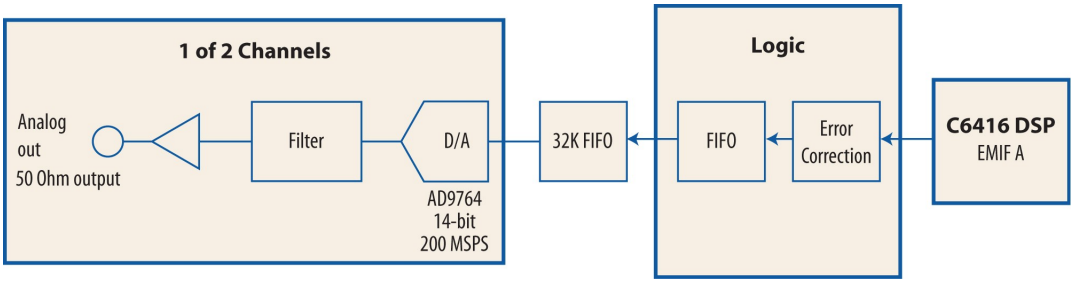


Figure 5. Simplified Quixote DAC channel

Timing and Synchronization Features

Quixote has many features to generate precision clocks and synchronization signals. An on-card precision, low-jitter programmable PLL provides clocks for either communications or FPGA clocks. External clock IO from each FPGA is provided to the front panel also. The clocks can also be shared to the PMC modules for sample rate generation through the FPGAs.

Synchronization signals for on-card or multi-card applications are provided for controlling data flows. Off-card synchronization signals are provided from each cluster FPGA. Synchronization signals between the FPGAs are also provided.

PCI Bus

The local PCI bus provides system-level connectivity to the on-card DSP, FPGAs and PMC resources over a local PCI bus architecture. The local bus is isolated from the host PCI bus by a PCI bridge chip to allow isolation of local traffic on the Quixote.

The following diagram shows a simplified view of the Quixote PCI architecture. (Only PCI buses are shown, not all connections, for clarity.)

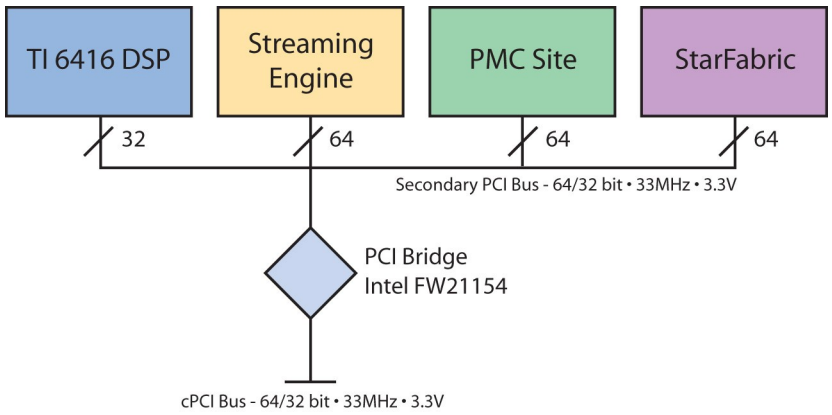


Figure 6. Simplified View of Quixote PCI Bus Architecture

One of the nice features on Quixote is the data streaming engine to PCI. This streaming engine provides high rate PCI communications to the processing core optimized for data streaming applications. Streaming can be achieved with minimal intervention by the DSP or host, saving computing power on each.

The '6416 also provides a flexible PCI interface for DMA to/from DSP memory. Software support for messaging and packet streaming is also provided.

StarFabric support, PICMG 2.17 compliant, is also available. System architectures that require high rate, deterministic communications can use StarFabric to provide flexible switched fabric architecture to their system design. On Quixote, the StarFabric may operate in PCI legacy mode to replace standard PCI or more advanced modes in cooperation with the PCI bus.

Digital Signal Processor

The Velocia baseboard's TMS320C416 DSP operates at 1 GHz and is a 32-bit fixed-point device. The DSP interfaces to the memory and peripherals on each baseboard through its external memory interface (EMIF), which has programmable definitions for the memory interface timing.

DSP External Memory

All Velocia baseboards provide 64 Mbytes of SDRAM memory mapped to the '6416 DSP EMIFA 64-bit memory space. This is the primary DSP memory for programs and data storage. On Quxiote, this memory runs at 100 MHz using an external clock for memory. Quadia memory runs at 133 MHz on EMIF A. The initialization of the memory space defines the correct parameters for the type of SDRAM used on the baseboard, including refresh timing, and should not be modified.

DSP Initialization

For proper operation of the external peripheral on the baseboard, the external memory interface control registers must be configured prior to use of the external memory interface. Applications built under the Pismo Toolset libraries will automatically initialize the registers appropriately (using code within HdwLib\IIInit.cpp). For those customers who need to initialize the registers manually, please refer to the EMIF register initialization values within the IIInit.cpp source file to obtain the required register values. Please note that the initialization is order sensitive and should be performed in the order given in the tables below.

Table 8. Quixote 'C6416 DSP EMIF Control Register Initialization Values

	Register Name	Address	Value	Use
EMIF A	EMIFA_GCTL	0x01800000	0x00012064	
	EMIFA_CE0	0x01800008	0x109103C1	Asynchronous devices
	EMIFA_CE1	0x01800004	0x000000E0	PCI FIFOs (burst)
	EMIFA_CE2	0x01800010	0x000000D0	SDRAM
	EMIFA_CE3	0x01800014	0xFFFFFF23	A/D and D/A FIFOs (burst)
	EMIFA_SDRAMTIM	0x0180001C	0x000005DC	
	EMIFA_SDRAMEXT	0x01800020	0x000D8DCB	
	EMIFA_SDRAMCTL	0x01800018	0x57338000	
	EMIFA_CE0SEC	0x01800048	0x00000002	
	EMIFA_CE1SEC	0x01800044	0x00000033	
	EMIFA_CE2SEC	0x01800050	0x00000002	
	EMIFA_CE3SEC	0x01800054	0x00000002	

	Register Name	Address	Value	Use
EMIF B	EMIFB_GCTL	0x01a80000	0x00012064	
	EMIFB_CE0	0x01a80008	0x000000B0	
	EMIFB_CE1	0x01a80004	0x4184C81C	Asynchronous devices
	EMIFB_CE2	0x01a80010	0x4184C80C	
	EMIFB_CE3	0x01a80014	0xFFFFFF23	
	EMIFB_SDRAMTIM	0x01a8001C	0x000005DC	
	EMIFB_SDRAMEXT	0x01a80020	0x000D8DCB	
	EMIFB_SDRAMCTL	0x01a80018	0x57338000	
	EMIFB_CE0SEC	0x01a80048	0x00000035	
	EMIFB_CE1SEC	0x01a80044	0x00000002	
	EMIFB_CE2SEC	0x01a80050	0x00000002	
	EMIFB_CE3SEC	0x01a80054	0x00000002	

The individual reset signals for each DSP target on the Quadia baseboard are controlled by the PCI bus. The normal boot sequence is to reset the DSP, load the application software, then pulse the PCI INT signal to launch the DSP from location zero. All Velocia family cards load the DSP software using the DSP PCI bus interface. Since the normal boot time of the PC is quite lengthy, the power supplies on the baseboard are stable before a valid software image can be downloaded to each DSP. Special precautions may need to be employed when interfacing external hardware to insure the target hardware remains benign during this startup interval.

Functions for loading the software and controlling reset are included in the Malibu Toolset. See the C64xDownload.exe applet description for details.

DSP JTAG Debugger Support

Standard TMS320 family JTAG debugger operation is supported by each Velocia baseboard. An external debugger connector allows use of industry standard JTAG debugger hardware from Innovative, Texas Instruments, and other third party suppliers. The DSP is the only device in the scan path for all cards except Quadia, which has the four DSPs in the scan path.

Software for JTAG debugging and code development is TI Code Composer Studio. The JTAG port in this case is used to control the DSP program execution and memory control.

Here are simplified views of the JTAG chains.

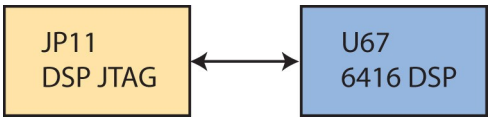


Figure 7. Quixote DSP JTAG Chain

See the appendix of this manual for the connector pinout, location and type information.

FPGA JTAG Support

The Velocia cards support FPGA debug over JTAG interface. Tools such as Xilinx Impact, ChipScope and SystemGenerator use the FPGA JTAG interface as their communications and control path to the FPGA. The connector is compatible is Xilinx debug tools such as Parallel Cable IV and most others.

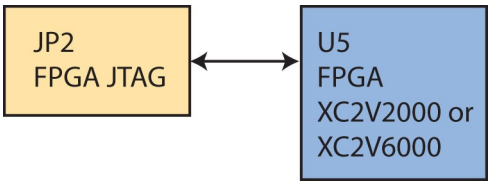


Figure 8. Quixote FPGA JTAG Chain

See the appendix of this manual for the connector pinout, location and type information.

Host/Target Communications

Overview

Many applications involve communication with the host CPU in some manner. All applications at a minimum must be reset and downloaded from the host, even if they run independently from the host after that.

Other applications need to interact with a host program during the lifetime of the program. This may vary from a small amount of information to acquiring large amounts of data. Some examples:

- Passing parameters to the program at start time
- Receiving progress information and results from the application.
- Passing updated parameters during the run of the program, such as the frequency and amplitude of a wave to be produced on the target.
- Receiving alert information from the target.
- Receiving snapshots of data from the target.
- Sending a sample waveform to be generated to the target.
- Receiving full rate data.
- Sending data to be streamed at full rate.

These different requirements require different levels of support to efficiently accomplish. The simplest method supported is performing file I/O from within Code Composer using either the standard C file functions (which communicate directly through CCS to the Host file system), or via the Innovative terminal emulator, which supports simple data input and control and the sending of text strings to the user in addition to file I/O.

On the Velocia family baseboards, the CPU Busmaster interface allows communication of command messages and data packets between target and host. The command packets are fully interrupt driven and allow about 16 words of data to be transferred in each packet. For full rate data transfers, the hardware supports block-oriented bus-mastering transfers supporting maximum-speed data movement between the target and host.

CPU Busmastering Interface

Each TI C64x DSP on the baseboard is capable of independent and autonomous PCI bus-mastering to move data between target and host memory. This bus-master facility can be used to transfer data between host and target applications. In addition, the interrupts are also used to support the exchange of Message packets consisting of 16 words of data. This allows the exchange of command and parametric data without involving bus-mastering.

CPU Busmastering Implementation

Packet Based Transfers

Some Innovative DSP boards, such as the those within the Matador family, feature streaming bus-mastering hardware in which (logically) data is an infinite stream between the source and destination. This model is more efficient because the signaling between the two parties in the transfer can be kept to a minimum and transfers can be buffered for maximum throughput. On the other hand this streaming model has relatively high latency when attempting to communicate asynchronous data blocks, since a data item may stall in internal buffering until subsequent data accumulates to allow for an efficient transfer.

By contrast, the CPU bus-master interface implemented within the the C64x DSP transfers discrete blocks between the source and destination. Each data buffer is transferred completely to the destination in a single operation. Only if several transfers are requested at once will any delay in beginning transmission occur, as multiple requests have to be serialized through a single hardware system.

The data buffers transferred can be of different sizes. Each requested buffer is interrogated for its size and fully transmitted. At the destination, the destination buffer is re-sized to allow the incoming data to fit. If the buffer given is too small for the data, it will be reallocated to allow the transfer. Reallocating buffers can take some time, for best performance buffers should be presized to be large enough for the largest transfer expected. This will make allocation of buffers at critical times unnecessary.

Blocking Interface

CPU bus-mastering uses a simple blocking interface for its send and receiving functions. The sending function will not return until the transfer has completed and the buffer is ready for reuse. Similarly, the receiving function waits until data has arrived from the data source and transferred into the data buffer before returning. At this point the buffer is ready for use. This blocking allows sequences of transfers managed by a simple sequence of calls to transfer functions.

Since the transfer functions are blocking, they are best avoided in the main user interface thread of a Windows application. The GUI will be appear to be ‘frozen’ until the transfer has completed. For best results, the data transfer functions should be placed in separate threads on the target and host applications. In fact, each direction of transfer should have its own thread, so that the two directions of transfer can interleave as much as possible. The example programs CpuBmIn and CpuBmOut illustrate the use of separate threads for data transfer.

Maximum Transfer Size

The largest transfer allowed is half of the total size of the DMA Buffer allocated by the INF file when the driver is installed. Half of the memory is dedicated to each direction. The default buffer size in the INF is 0x200000 bytes, so the maximum transfer is 1 Megabyte.

Malibu Library Host Support for CPU Busmastering

In concept, there are a large number of ways that data can flow data between PCI resources. Data can be be bus-mastered or slave accesses can be used. When bus-mastering, data can flow continuously, referred to as streaming, or intermittently. Consequently, Malibu has been designed such that baseboard objects such are Quadia or C64xDsp do not contain embedded support for data flow via the PCI bus. Rather, in order to isolate encapsulate the details of particular bus-mastering strategies and provide a means by which baseboards can perform the type of data flow most appropriate for each application, baseboard objects must be logically connected to an independent communications object which is responsible for communicating data to and from Host memory. Logically, a baseboard is *connected* to a communications object which implements a particular communications strategy.

The TiBusmasterStream object is one such communications object. Applications instantiate an object of this type, then associate it with a baseboard object in order to allow it to perform communications functions. The TiBusmasterStream::ConnectTo() method is used to establish this association. Once connected in this fashion, the Send() and Recv() methods may be used to transfer buffers of data between the Host CPU and DSP baseboard.

```
//
//  Block Transfer System Methods
virtual bool  Send( const IntegerBuffer & packet );
virtual bool  Recv( IntegerBuffer & packet );
```

`TiBusmasterStream::Send()` sends the contents of a `IntegerBuffer` object to the target. All of the data in the buffer is transferred. There is no means of sending a partial buffer. The function will not return until the entire block has been transferred to the recipient DSP.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

`TiBusmasterStream::Recv()` waits for data to arrive from the target, then returns the data in the buffer provided. The `IntegerBuffer` buffer will automatically be re-sized to fit the data transferred from the source. If the buffer is smaller than the amount of data received, this may involve a reallocation of the data block.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

Packet Notification Events

The TiBusmasterStream object contains an event that will be signaled when a packet buffer arrives from the target. This OnPacketAvailable event can have a handler installed that will process the message, thus eliminating the need for a separate thread to manage incoming data packets. The Recv() method can be called in the handler with the assurance that the request will not block, since data is already present.

Target (Pismo Library) Support for CPU Busmastering

In the Pismo library, the UtilLib library contains a file PciTransfer.h that contains this class:

```
class PciTransfer : public PciTransferBase
{
public:
        PciTransfer();

        bool  Send(int channel, const Buffer & buffer);
        bool  Recv(int channel, Buffer & Buffer);
```

```
        ...
};
```

`PciTransfer::Send()` sends the contents of a Buffer-derived object to the Host. All of the data in the buffer is transferred. There is no means of sending a partial buffer. The function will not return until the block has been transferred to the host.

The use of the base buffer class allows any of the `IntBuffer, CharBuffer, FloatBuffer` and similar classes to be sent across the interface.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

`PciTransfer::Recv()` waits for data to arrive from the target, then returns the data in the buffer provided. The Buffer will be re-sized to fit the data transferred from the source. If the buffer is too small, this may involve a reallocation of the data block.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

Packetized Message Interface

The C64x processor's PCI interface is also used to support a lower bandwidth communication link for sending commands or parametric information between target and host. Library support is provided to build a packet-based message system between the target and host software. These packets can provide a simple yet powerful means of sending commands and information across the link between the two processes.

Message Mailbox Emulation

On Matador baseboards, a set of sixteen mailboxes in each direction to and from the host PC are shared with the DSP to allow for an efficient message mechanism that complements the bus-mastering interface. These mailboxes have a handshake mechanism that signals the recipient for the availability of data, and a corresponding signaling to the sender when the message was received.

The C64x processor has no such facility, but the software has been designed to emulate the same interface. Message Packets of 16 words are cached in a memory region that can be accessed by the host when signaled that data is available. Interrupts are used for signaling in both directions, allowing for rapid response. But since the packets themselves are small, throughput is relatively modest. The packet interface with bus mastering is preferred for large volumes of data.

Software in the Pismo Toolset and Malibu implement a message system that allows the application programmer to use the mailbox system for command and control, lower rate data passing, and status queries and replies, as well as many other uses.

The Message System

The Message system provides a single bi-directional link between the target and the host. More complicated arrangements are possible by using the messages to encode data sources and destinations. Two words of the message are dedicated to this kind

of routing information, with three fields defined for routing or message type encoding. The remaining space can be used for data.

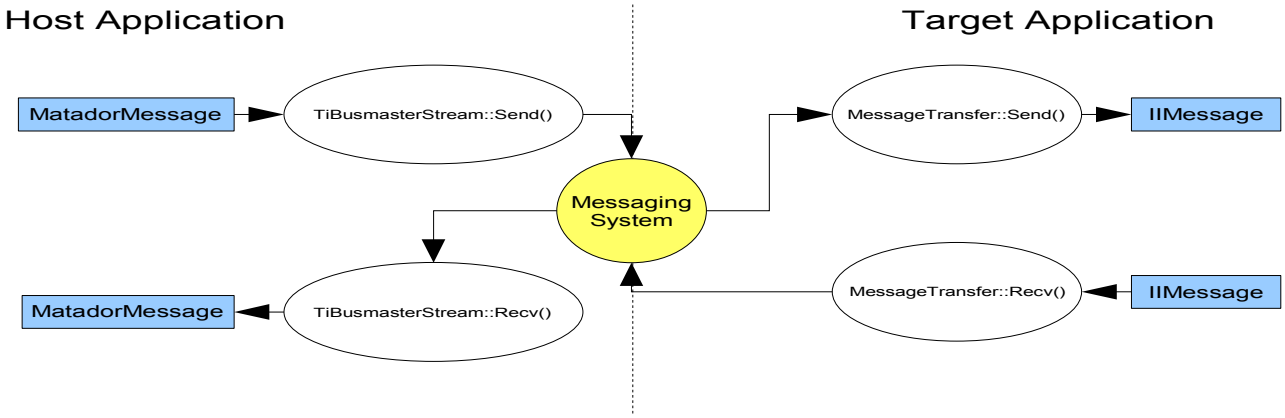


Figure 9. Messaging System Objects

Host-side Message Objects

Messages consist of packets that may contain up to 14, 32-bit data words plus two 32-bit header words. The details of packet formatting are hidden on both the target and the host by the use of similar Message objects to encapsulate the packet to be transmitted. On the host side this message packet class is called `MatadorMessage`. On the target, the corresponding class is called `IIMessage`.

Messages sent by the target are collected into `MatadorMessage` objects for delivery to the event handlers dedicated to responding to the messages. For all practical purposes, you can think of the Message System as exchanging `IIMessage/MatadorMessage` objects.

The header portion of the Message Packet contains some system data and some fields that can be used by the application. The `TIIMessage` Object provides properties to access these fields:

Table 9. TIIMessage Header Field Access

Channel	Property	Message Channel (Free for use in application).
TypeCode	Property	Message or Command Type
MessageId	Property	Message counter or other user data
IsReplyExpected	Property	Set if reply is needed. Free for use in application.

The 14 words of Data are accessible as arrays. Array properties are defined to allow loading common data types into a message:

Table 10. TIIMessage Data Section Interface

Data[]	Property	Access the data region as 32-bit integers (0-13)
AsFloat[]	Property	Access the data region as floating point data. (0-13)
AsShort[]	Property	Access the data region as 16-bit integers. (0-27)

Data[]	Property	Access the data region as 32-bit integers (0-13)
AsChar[]	Property	Access the data region as 8-bit characters. (0-55)

Message Packets supporting a mix of data formats are supported as long as the user remembers to make sure the individual portions do not collide in the message data. For example when mixing a **float**, a **char** and a **short** and an **int** (in that order) the index of the float is 0, the char is at char-index 4, the short is at short-index 4, and the integer can go at integer index 2. Creating a wrapper class to handle the indexes can make the use of mixed-mode indexes transparent to the user. Failing that, using only 32 bit wide data for floats and integer types makes the indexing clear, at the expense of data packing efficiency.

Target Side Message Objects

On the target side, the Pismo library supports a very similar class, `IIMessage`, to contain the message. However, since on the target Properties are not supported, we instead use a convention to define setter and getter functions in a uniform manner. These paired methods are called "a property" or "property methods". (See the Pismo Online Help under 'property' for more details). The convention is to overload the method name, distinguishing getter function by a 'const' declaration and the setter by an additional argument for the value to set. For example, consider the abridged definition of the ClockBase class below:

```
class ClockBase : public CslNoncopyable
{
public:

    ...

    // Enabled is a setter/getter property
    void            Enabled(bool state);
    bool            Enabled() const;

    ...
};
```

The following table gives the header field access methods for the class::

Table 11. IIMessage Header Field Access

Channel()	Property Methods	Message Channel (Free for use in application)
TypeCode()	Property Methods	Message or Command Type
MessageId()	Property Methods	Message counter or other user data
IsReplyExpected()	Property Methods	Set if reply is needed. Free for use in application.

The 14 words of Data are accessible as array property methods. These methods all have an additional argument giving the index into the data section:

Table 12. IIMessage Data Section Interface

Data()	Property Methods	Access the data region as 32-bit integers (0-13)
AsFloat()	Property Methods	Access the data region as floating point data. (0-13)

Data()	Property Methods	Access the data region as 32-bit integers (0-13)
AsShort()	Property Methods	Access the data region as 16-bit integers. (0-27)
AsChar()	Property Methods	Access the data region as 8-bit characters. (0-55)

Message Packets supporting a mix of data formats are supported just as on the host side.

Message Communication

The packetized message system is event driven. When the sender posts a message packet, at the first available opportunity the packet is loaded into a reserved memory region and an interrupt is generated to the receiving side. On the receiver, the interrupt is detected and the application thread waiting for the messages is notified. After processing the message, the sender receives an acknowledgment that the previous packet has been processed and the memory region is free for another transmission. The application at the same time processes the message data and performs whatever action is needed.

To establish a bi-directional link you need a sender and a receiver on both the target and the host.

The 14 words of `Data` are accessible as array property methods. These methods all have an additional argument giving the index into the data section:

Table 13. Message Sending and Receiving Methods

Direction	Sender	Sender Type	Receiver	Receiver Type
Target to Host	MessageTransfer::Send()	Member Function	TiBusmasterStream::Recv()	Blocking call
Host to Target	TiBusmasterStream::Send( )	Member Function	MessageTransfer::Recv()	Blocking call

Before using messages, the target and the host must instantiate threads which are responsible for receiving and sending all message traffic. Within these threads, the user code will block in any call to Send or Recv until the message has been communicated from the sender to the receiver. Consequently, these calls must not be made from within the main thread within Windows applications, since that thread must respond to system messages.

On the host side, you can use events to provide notification of the arrival of a message. If you provide a handler for the OnMailAvailable event in the TiBusmasterStream object, the handler will be called on the arrival of a new mail message. This notification can be synchronized with the main GUI thread to allow updates to the user interface. Since the message has already arrived, calling Recv() inside the event handler will not cause the main GUI thread to block.

The messaging system uses the same interrupt subsystem as does the bus mastering interface. Therefore, messages may be transferred in either direction while data streaming is in progress, but bus-mastering transfers and messages will be serialized. Under normal circumstances, there may be some minimal bus-mastering speed degradation due to the increased load, but given that the messaging system is only designed for moderate bandwidth communication this should not be significant. Since the signaling of data available and the acknowledgment require interrupts, it is a requirement that global interrupts be enabled for messages to proceed.

C++ Terminal I/O

The terminal emulator applet is a Host PC application which provides a C++ language-compatible, terminal emulation facility for interacting with the TermIo Pismo library running on an Innovative Integration DSP processor.

Using the terminal emulator, it is possible to develop and debug target DSP code while deferring development in Host application code. By using simple, streaming I/O functions within a target application during development, DSP algorithms can be developed independently from Host applications. Later, when a custom Host application code is written, the DSP standard I/O functions may be deleted from the target application and the target application will no longer be dependent on the emulator or the target TermIo libraries.

Streaming methods such as << and >> are dispatched by the TermIo object to route text and data between the DSP target and the Host terminal terminal emulator applet. Text strings are presented to the user via a terminal emulation window and host key-board input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation.

Target Software

All of the features of the terminal are accessed through the two classes TermIo and TermFile. TermIo provides the basic streaming interface which allows text messages to be formatted and streamed out to the terminal as well streaming in strings and numeric values from the terminal for consumption by target application code. The TermFile class provides a mechanism allowing target applications to open host disk files, perform read and write accesses, and subsequently close these files. See the Files.cpp example for illustrative usage of each of these classes and their functions.

Tutorial

Using the terminal during target software development is simple. The global cio object which is automatically instantiated within the Pismo libraries. Use the methods within the cio class to format text strings and then stream them to the Terminal applet

```
cio << bold << "7Demonstrate file I/O\n\n" << normal << endl;
```

Note the use of manipulators, such as bold and normal, to force formatting of the text string as it is streamed to the host. TermIo features many such manipulators to perform functions such as setting text color (setcolor), clearing to end-of-line (clreol), clearing the screen (cls) and so forth. Other manipulators are available to format numeric values as they are streamed to the host. For example, the phrase

```
cio << "Hello" << hex << showbase << 4660 << dec;
```

displays the string "Hello 0x1234" on the console display, converting the integer value 4660 as a hexadecimal number on the target, prior to streaming it to the host. Other manipulators are available providing extensive control over the display of floating point numbers as well as integer values.

It is also frequently necessary to obtain input from an operator during the run-time execution of a target application. For example, it may be necessary to prompt for a sample rate at which analog I/O is to be streamed. The code fragment below illustrates the necessary technique

```
// Prompt the user
cio << " Enter a float: " << flush;
float x2;
// Eat user input
cio >> x2;
```

The stream manipulator >> is overloaded to allow streaming directly into floating point, integer and string variables directly from UniTerminal.

To perform file input and output from within target applications, first instantiate a TermFile object, as below

```
TermFile File;
```

Then, use the TermFile Open method to open the file for access on the host using the desired open attributes

```
if (!File.Open("wave.bin", "w+b"))
    {
    cio << "nOutput file open error - Program terminating!" << endl;
    cio.monitor();
    }
```

This method returns a Boolean indicating success if the file open is successful. To store data into the file or retrieve data from the file, use the Write or Read methods, respectively. For example

```
transferred = File.Write((char*)&Buffer[0], 10000);
```

writes 10000 bytes of Buffer into the disk file. When disk operations have been completed, the file should be closed using the `TermFile::Close` method.

Programming with the Pismo Class Library

In order to support the baseboard as a part of a complete system, a complete set of powerful software libraries is provided to program the DSP on the baseboard and also to allow the card to interact with a host program resident on the PC. The Pismo Class Library provides support for developing applications which run on the target baseboard. The Malibu Library provides the library support for host application development.

Pismo provides extensive C++ class support for:

- Dynamic creation and runtime control of tasks
- Simplified management of and access to all TI Chip Support Library (CSL) and DSP/BIOS API functions including: Semaphores, Mutexes, Mailboxes, Timers, Edma, Qdma, Atoms, McBsp, Timebases, Counters, etc.
- Data exchange using RTDX Streaming I/O
- Foundation (base) classes for DMA-driven device driver development
- Templatized queues
- Partial standard-template library functionality via STLPort

For example, the code fragment below uses the Pismo IntBuffer class to initialize a QDMA (quick DMA) to perform a memory-to-memory move of a constant value (0) into a 4096-word buffer (at Src), then to copy the source buffer (Src) to the destination buffer (Dst):

```
// Create a source buffer of 0x1000 integers
IIBuffer Src(0x1000);
// Initialize the source buffer with zeros
Src.Set(0);
// Create a destination buffer of 0x1000 integers
IIBuffer Dst(0x1000);
Dst.Copy(Src);
```

Simple To Use
In the same way, peripheral-specific class libraries dramatically simplify access to board-specific peripheral features. For example, the code fragment below illustrates use of the PCI communications library functions to send a buffer of data to the Host PC using bus-mastering.

```
PciTransfer Xfer;

const int Size = 0x10000;
IntBuffer Buffer(Size);
for (int i = 0; i < Size; ++i)
    Buffer[i] = i;
//
//  Transmit buffer back to host
```

```
Xfer.Send(0, Buffer);
```

Not Just for C++ Experts
Note that even if you're not a C++ maven, the code is quite clear and understandable. In fact, one of the benefits of using C++ is that while it helps to mitigate and manage complexity to support creation of larger, more sophisticated applications, it is often simply used as a "better" dialect of the C language. C++ is essentially a superset of C. As such, you may freely intermix calls to legacy 'C' functions, newly-written C functions, Assembler functions and C++ functions (called methods) within C++ programs. You need not fully understand all of the enhanced capabilities and features of C++ in order to fully exploit the features of the class libraries provided in Pismo.

Unique Feature Support for each Baseboard
The Pismo Library for each baseboard provides classes and functions to access the unique features of each baseboard. For example, the Quixote version provides device drivers for AnalogIn and AnalogOut to acquire data from the analog hardware. The Pismo software isolates the application programmer from the complexities of both the hardware and DSP/BIOS.

DSP Programming on the Baseboard

Innovative Integration's *Pismo* is a software suite allows the developer to fully exploit the advanced hardware features of the Innovative Velocia DSP product line and to reap all the benefits from DSP/BIOS. Every board peripheral has been carefully integrated into the OS and its functionality encapsulated in a device driver that can readily be controlled within DSP/BIOS applications including PCI interface, analog I/O, external bus and memory, serial ports and other I/O devices.

These drivers expose all the necessary parameters needed to efficiently control each function of the peripherals. Any peripheral board resource may be instantiated, configured and shared among program tasks. The device drivers also take care of assigning default values for unspecified or non-critical parameters of a function. C++ is used as the foundation for the Pismo libraries, but C programmers may use Pismo freely, without having to learn C++ details or C++ extensions to the C language. The C++ libraries provided in Pismo are far more capable, complete and easy-to-use than any previous generation of DSP peripheral support libraries from Innovative Integration.

Illustrative, real-time example programs are included in the software suite along with complete project files and DSP/BIOS modules. The examples act as a springboard for the development of custom, high-performance DSP application programs.

Device Drivers

Most of the peripheral devices supported by Pismo under DSP/BIOS are accessed and controlled via custom, DSP/BIOS-compliant device drivers. These drivers, provided by Innovative Integration with each DSP baseboard, are installed as "User-Defined Devices" and are located under Input/Output | SIO - Streaming Input and Output Manager | SIO Drivers within DSP/BIOS CDB files. Drivers for each supported peripheral are pre-installed in the board-specific CDB template file provided in the Developers Package for each DSP baseboard.

Advantages of using DSP/BIOS drivers
By providing DSP/BIOS drivers for high-speed data flow between application code and peripheral devices, application programs may be developed without requiring application programmers to have detailed knowledge of the underlying peripheral hardware present on the DSP baseboard. Rather, to initialize peripherals and flow data, applications make use the stock DSP/BIOS API SIO function calls which are encapsulated within the Pismo wrapper class, Stream.

How to use a DSP/BIOS driver
Before using a DSP/BIOS driver, it must be opened for use via the `Stream::Open` method. Afterwards, the `Stream::Control` method may be used to perform any necessary device-specific initialization and/or control functions. Buffers of data are then efficiently exchanged between application and driver code via the Stream class methods `Stream::Put` and `Stream::Get` (or their lower-level sister functions Issue and Reclaim). These methods are efficient because they effect data flow via buffer pointer manipulation instead of expensive data copy operations. After use of a device is complete, it is closed using the `Stream::Close` method.

In Pismo, several input and output classes derive from Stream, to simplify and standardize access to features of each the streaming device driver. For example, the analog input and output drivers are controlled from the AdcStream and DacStream objects. Similarly, the PCI input and output drivers are controlled from PciInStream and PciOutStream. Use these objects where available, in preference to the more basic Stream base class, since the derived objects encapsulate and hide the complexity of controlling the analog, digital and PCI hardware.

The example below illustrates use of each of the above methods to open the drivers for the audio input and output codecs on the Toro DSP baseboard, set the sampling rate, echo the signals received on the analog inputs to the analog outputs, then close the driver.

```
// Desired rate of buffer cycling during streaming
const float BuffersPerSec = 10;

//---------------------------------------------------------------------------
//  IIMain() --  Loop analog input to output
//---------------------------------------------------------------------------

void IIMain()
{
        volatile bool run = true;
        volatile bool status;
        float SampleRate;

        //
        //  Terminal I/O
        cio << init;
        cio.At(Point(25, 0));
        cio << bold << "\7Echo Application\n\n" << normal << endl;

        cio << "Enter the sample rate (Hz): " << flush;
        cio >> SampleRate;
        cio << "\n  You entered: " << SampleRate << endl;

        // Instantiate the analog stream objects
        AdcStream Ain;
        DacStream Aout;

        // Simple, continuous data flow
        BasicTmb Timebase;
        Timebase.Rate(SampleRate);
```

```
        Ain.Device().Attach(Timebase);
        Aout.Device().Attach(Timebase);

        int MaxChannels = std::min(HardwareInfo()->AdcChannels(), HardwareInfo()->DacChannels());
        int EventsPerBuffer = SampleRate/BuffersPerSec;

        // Enable all analog input and output channels
        for (int i = 0; i < MaxChannels; ++i)
                Ain.Device().Channels().Enabled(i, true);
        // Size the stream buffers to signal at specified rate
        Ain.Events(EventsPerBuffer);

        for (int i = 0; i < MaxChannels; ++i)
                Aout.Device().Channels().Enabled(i, true);
        // Size the stream buffers to signal at specified rate
        Aout.Events(EventsPerBuffer);

        status = Aout.Open();
        status = Ain.Open();

        // echo input to output
        cio << "\nEchoing A/D to D/A... at " << SampleRate << " Hz\n\n" << endl;

        cio << noshowcursor << flush;
        int Count = 0;
        while(!cio.KbdHit())
                {
                cio << "\rPlaying buffer: " << ++Count << flush;

                Ain.Get();
                Aout.Put(Ain.Buffer());
                }
        cio << showcursor << flush;

        // Terminate streaming
        status = Ain.Close();
        status = Aout.Close();

        cio << "\n\nProgram terminated." << endl;
        cio.monitor();
}
```

In the example above, two device drivers are involved as data flows from the Ain device to the Aout device. Specifically, Ain and Aout are custom drivers, provided by Innovative Integration to drive the A/D and D/A devices present on the DSP board. The input device driver is named AnalogIn and the output device driver is named AnalogOut. In the example above, the analog chain configured to operate at a user-entered sample rate to flow all samples acquired from all input channels to all output channels.

Driver-specific control functions

Some Pismo drivers support special `Stream::Control` methods used to configure a device driver for a particular mode of operation, data format or other configuration or control operation outside of the scope of simple data flow. While these special `Control` methods ma be called directly, their syntax is akward because the `Control` method does not preserve type information as it conveys data into the device driver. The wrapper classes, such as `AdcStream` and `DacStream` provide type-safe, easy-to-use methods which access to all supported, underlying `Control` functions. support control Details on available control methods for each specific driver provided in the Pismo toolset are provided in the online help files.

Driver Buffer Model
Each device driver, when opened, allocates buffers of a user-specified size (BufSize, in the example above) to be used as the destination for data samples accumulated during signal input or as the source for data samples consumed during a signal output.

DSP/BIOS device drivers implement data flow through a buffer passing mechanism. In the example above, analog sample data is continuously routed from the Ain to the Aout device via the code fragment

```
Ain.Get();
Aout.Put(Ain.Buffer());
```

which causes each of the buffers read from the `Ain` device to consumed by the `Aout` device. This is accomplished by successively passing the pointer to the data buffer most recently filled by the `Ain` driver directly to the `Aout` driver, without copying the contents of the buffer.

By default, the `Ain` and `Aout` devices are each allocated two internal buffers. Additionally, each stream object implicitly allocates one additional buffer for use in the buffer pool. Thus, each of the drivers illustrated above is allocated a total of three buffers which are managed as a rotating pool, by DSP/BIOS. If desired, the number of offers present in the pool may be modified prior to opening the driver by assigning a new value using the BufferCount method of the wrapper objects. For example,

```
// Instantiate the analog stream objects
DacStream Aout;
Aout.BufferCount(5);
AdcStream Ain;
Ain.BufferCount(5);
```

would force DSP/BIOS to allocate five internal buffers for each stream which, when combined with the single buffer implicitly allocated with each Stream object, would result in a total of six buffers in the DSP/BIOS managed pool for each device driver.

The size of these buffers may be specified explicitly, using the `Stream::BufferSize` method, or automatically calculated using the `AdcStream/DacStream::Events` method. This latter method sizes the buffers used by the streaming device driver such that they can contain the specified number of acquisition “events”, where an event is defined as one sample from all enabled A/D or D/A channels. This simplifies most buffer processing algorithms since all buffers are guaranteed to contain an integral number of samples from all enabled channels.

Generally, more buffers in the driver pool results in greater instantaneous load-carrying capacity. In practice, a larger number of pool buffers equates to a longer duration of time over which the application program can safely neglect the data servicing requirements of the device driver without risking data integrity errors. For example, in the example above the originally allocated three buffers per driver, each sized at 0x1000 bytes, running at 44.1 kHz equates to a load carrying capacity of

```
(0x1000 bytes/buffer) x (3 buffers) / (44100 samples/sec) /
(2 bytes/sample) = 139 mS
```

Whereas in the second example, with six buffers per driver pool

```
(0x1000 bytes/buffer) x (6 buffers) / (44100 samples/sec) /
(2 bytes/sample) = 278 mS
```

So, in the first example is the application program were to become busy and momentarily neglect service servicing the Ain and Aout devices for > 139 milliseconds, data integrity would be compromised and the analog output would not track the

sine wave generated by the Ain driver. However, in the six-buffer example, which provides greater instantaneous load carrying capacity, data integrity would be preserved at the expense of additional memory utilization.

Driver Types

While all device drivers provided in the Pismo toolset are DSP/BIOS-compliant and accessible via the Stream class as illustrated above, there are two distinct categories of DSP/BIOS device drivers implemented within Pismo - continuous and burst.

Continuous drivers are implemented for peripheral devices which, during operation, may utilize a continuous conversion clock. Devices which fall into this category are A/Ds, D/As and codecs. Drivers written for devices of this type must be capable of sustaining continuous data flow. Special provisions may be made in specific drivers to support obtaining snapshots of the data which is flowing non-continuously, but the default mode of operation involves continuous, uninterrupted dataflow, and the driver must be capable of supporting this. For example, data which flows between an A/D converter and a buffer on the target DSP must not be suspended or inhibited for greater than one conversion sample interval, or else data loss will result and the resultant captured waveform will appear distorted on close inspection.

Burst drivers are implemented for peripheral devices which do not generate or consume data continuously during operation. Devices which fall into this category are the PCI bus interface, the FifoPort or any other device which is capable of pacing real-time dataflow through some form of handshake mechanism. For example, data which flows between the host PC and the target DSP via the PCI bus may be sent at irregular intervals and irregular packet sizes. When either the host PC or the target DSP becomes momentarily busy and unable to exchange data via the PCI bus, dataflow is temporarily paused. However, there is no risk of data loss since the bus interface logic on the DSP baseboard provides a hardware handshake interlock which paces dataflow until both the PC and DSP baseboard are again ready for the data exchange to resume. Although dataflow is burst, aggregate data transfer rates may still be very high for burst drivers. The primary distinction is that data does not flow continuously at regular sample intervals as is the case with continuous drivers.

Pismo provides C++ base classes to enable creation of either of these driver types. Continuous drivers derive from the SioDaxDriver and SioDaxDma base classes. Burst drivers derive from the SioDriver, SioDev and SioDma base classes. Both types of drivers rely heavily on the DMA hardware present in the DSP to perform high-speed data acquisition and signal generation functions efficiently.

Driver Implementation

Both continuous and burst drivers are implemented using DMA as the data movement mechanism. The use of DMA ensures data integrity throughout the lifetime of the streaming operation. Additionally, the DMA reduces the rate of interrupt servicing by the CPU, resulting in optimal CPU bandwidth preservation.

Applications using a continuous driver are expected to service the buffers in the pool associated with that driver in real-time. That is, within the constraints of the load carrying capacity afforded by the internal buffer pool associated with the continuous driver, application code is expected to consume data from an input device or provide data to an output device at rate nominally >= the sample rate of the conversion clock being used to drive the underlying peripheral associated with the continuous device driver.

Applications using a burst driver need not service the buffers in the pool associated with that driver in real-time. While protracted neglect in servicing the buffers in the pool associated with the driver will result in degraded throughput, data integrity is never at risk when using a burst-style driver.

DMA-enabled Drivers
While the above benchmarks are impressive, it's important to realize that the features and facilities of BIOS, used in conjunction with the silicon enhancements available in the C6000 DSPs, make DSP/BIOS-based DSP-applications less sensitive to the performance of some of these operations. For example, a combined 0.45 uS context save and restore for a hardware interrupt is certainly state-of-the-art. Hardware interrupt timings such as this are often used as a yardstick to measure the real-time performance of an embedded operating system.

But the DSP/BIOS-compliant device drivers provided by Innovative in the Pismo package fully exploit the available DMA channels in the C6000 DSPs so that hardware interrupt rates rarely exceed one KHz! The net effect is that virtually all of the bandwidth of the CPU is available for application processing. The CPU bandwidth consumed by the Innovative-supplied DSP/BIOS device drivers is minimal. Therefore, the relative importance of blazingly-fast hardware interrupt response times is decreased.

Simplified Use
Due to the relative complexity involved in programming DSP DMA channels compared to using CPU interrupt handlers for data movement, most application programmers simply avoid use of DMA entirely, resulting in highly inefficient use of CPU computational resources. In providing DSP/BIOS-compliant device drivers for all real-time peripherals within Pismo, the application programs may focus exclusively on the end-application, rather than the myriad details involved in peripheral setup, initialization and servicing. Further, use of the Pismo driver insures maximal CPU availability for application use.

For example, consider the code fragment below which illustrates all of the steps necessary to fully initialize and stream a sine wave to the audio output codec present on the Innovative Vista board at 44.1 KHz:

```
//-------------------------------------------------------------------------
//  IIMain() --  Generate a waveform on both audio channels
//-------------------------------------------------------------------------

void IIMain()
{
        const int BufSize = 0x1000;
        volatile bool status;

        // Basic I/O...
        cio << init;
        cio.At(Point(35, 0));
        cio << bold << "\7Waveform Generation Demo\n\n" << normal << endl;

        // Instantiate the analog stream objects
        DacStream Aout;
        Aout.Name("/AudioTee/AnalogOut");
        Aout.BufferSize(BufSize);
        Stream Ain("/SineAin", smInput, BufSize);

        // Open the analog I/O drivers
        status = Aout.Open();
        status = Ain.Open();

        // Start the Dds
        Aout.Control(dcSetSampleRate, 44100);

        cio << "Streaming 1 kHz sine wave to audio output driver..."<< endl;

        int Count = 0;
        // Generate waveform, send to D/As
        while(!cio.KbdHit())
                {
                if (!(++Count & 0x0f))
```

```
                        cio << "\rBuffer: " << Count << flush;

                Aout.PutFrom(Ain);
                }

        // Close the analog I/O drivers
        Ain.Close();
        Aout.Close();

        cio << "\nStreaming terminated."<< endl;
}
```

Multitasking Friendly
In addition to minimal processor loading, automatic DMA configuration and optimal bus utilization, the Pismo DSP/BIOS drivers support efficient cooperation in multitasking applications. For example, in the code fragment above, the call to PutFrom within the IIMain function will efficiently block until data is available from the Ain streaming device, allowing other tasks within the application to execute.

Analog Timebase Objects

Timebase objects provide a means to collectively configure a clock source, a start trigger and a stop trigger to control the baseboard logic which is used to pace and store the conversions of baseboard analog or digital peripherals. Timebases may thought of as external, independent physical devices like a precision oscillator timebase with programmable start/stop enables. In reality, they control one or more physical resources located on the Matador DSP baseboard. However, this portrayal of the timebase as a “virtual” clock source has advantages: For example, the Conejo baseboard contains six programmable timebases, each with different resolutions and capabilities. Which timer should be used for driving a sigma-delta converter? How are they configured when externally gating? The timebase components conceal the complexities of timebase programming by providing a separate component for each clocking technique or mode, so that you may remain blissfully-ignorant of low-level timebase initialization, routing and control mechanics.

Timebase Usage
Timebases are logical extensions to the Pismo streaming device drivers,such as AnalogIn and AnalogOut. As a stream object is created, used and finally destroyed within a target application, it performs specific driver/timebase operations at specific times. Each timebase object provides four virtual methods which are called during the lifetime of a stream object: Configure, Start, Stop and Unconfigure. These methods are perform timebase-specific initialization, and trigger functions that a stream driver automatically calls as the stream object is used. This way, application programs can be assured that these critical timebase functions are performed in the proper order and at the right time during program execution, withoout having to carefully code these operations within applications directly. These operations are summarized in the following table.

Table 14. Timebase Operations

Stream Operation	Timebase Operation
Attach	Current timebase configuration is copied into a dynamically-created timebase object for exclusive use by driver
Open	Driver timebase is Configured - clocks routed, triggers initialized

Stream Operation	Timebase Operation
Put/Get	Driver timebase is Started. Device-specific options are applied immediately prior to initiating data flow. For boards incorporating sigma-delta converters (Delfin), this includes application of AdcModeCtl and DacModeCtl options settings
Idle	Driver timebase is Stopped
Close	Driver timebase is Unconfigured

For more information on timebase options and configuration, see the chapter on Analog input and output.

Interrupt Handling

In DSP/BIOS, all hardware interrupts are intended to be managed by a DSP/BIOS hardware manager. This manager allows user functions to be called as part of the interrupt process while still cooperating with DSP/BIOS. As a part of the configuration process, the user can direct the HWI manager to call a user function.

Interrupts in a C++ Environment

In a system using C++, this means of attaching interrupts leads to several difficulties. A minor problem is that of name-mangling. C++ creates a new name for every function created in order to allow overloaded functions. The DSP/BIOS configuration does not understand the new name and results in a linker error. There is a simple work-around for this:

```
extern "C"
{
    void  MyHandlerFunction( void * arg );
}
```

This declares to the compiler to create a standard C symbol name for this function `(_MyHandlerFunction)` which can be used by to the DSP/BIOS configuration tool.

A more fundamental problem is that this mechanism does not allow the interrupt handling function to be changed during the life of the program. Also, this handler function may not be a class member function. This restriction can make designing a class object that handles interrupts awkward.

The Pismo Solution

The solution implemented in the Pismo environment is to take over all interrupt handling by providing a full set of standard handlers. The user then never needs to work in the CDB editor to provide handlers. The standard Pismo handlers contain code that will call a user's installed interrupt handler function if one is provided. While this adds a small amount of latency to the interrupt, the DSP/BIOS overhead per interrupt call is still much greater and dominates the total time per interrupt.. In general, the BIOS environment is not suited for extremely high interrupt rates. Luckily, the use of DMA to aquire data from FIFOs on peripherals means that high rate interrupt handlers are not needed.

Pismo uses a special object, a Binder, to group a handler function and its arguments in a way that can be properly called by the standard handler. One form of Binder is used to attach a stand-alone function and its arguments, another form allows the binding of an Object, a member function of that object, and its arguments. This form of binder can allow a class object instance variable to act as a handler for interrupts. Here is an example from the **Messages** example of defining a binder for a timer interrupt:

```
//
//  Timer Interrupt Handler Function
void  OnTimerFired(int arg);
//
//  Binder Object for Timer
typedef void (*IntFtnType)( int arg );

FunctionHandler<IntFtnType, int> TimerBinder(OnTimerFired, 0);
```

And attaching the binder to an interrupt:

```
//  Set up a real time clock to send commands to host on
//    Target channel...
Irq Timer0( intTimer0 );
Timer0.Install( TimerBinder );
Timer0.Enable( false );
//
//  Turn on the clock at 5 hz
DspClock  Tclk0(50.0, 150.0);

Timer0.Enable( true );
```

In the example, `TimerBinder` is an object that collects the handler function, `OnTimerFired`, and its argument, 0. This object is passed into an `Irq` object associated with the `TCLK0` interrupt. When the timer interrupt fires, the handler will be called with its argument. The binder is a template, allowing any type of argument to be used with an interrupt handler.

Class Irq

Class `Irq` is an object that can be created to manage a specific interrupt. It has functions to set, clear, enable and disable the interrupt and also allows a handler to be installed that will be called whenever the interrupt fires. In the above code, see how all functions involving the interrupt were encapsulated in the methods of the `Timer0` class object.

Interrupt Lock Classes

A common need in a program is the ability to disable a particular interrupt, or all interrupts, in a portion of the program. The standard means of standalone functions (an disable followed by a enable interrupts) has a few problems. The first is that the means does not nest well. If a function blocking interrupts is nested in a second one, interrupts will be re-enabled at the wrong time. A second is that if the function has multiple return paths, each must have the re-enable code in it. The introduction of C++ exceptions makes this problem even worse.

The Pismo library provides a set of class objects that meet this problem. These lock objects disable a particular interrupt or all interrupts in a region and restore the state to what it was on entry when the lock object is destroyed. If the object is created on the stack, any means of exiting the block in which the object is defined will cause the cleanup code to be called. Calls to these objects properly nest as well.

Table 15. Interrupt Lock Classes

Lock Class	Interrupts Affected	TI Class Library
InterruptLock	One IRQ	CSL.
GlobalIntLock	All interrupts	CSL.
HwiGlobalIntLock	All interrupts	DSP/BIOS.

Interrupt Binder Templates

The Binder system can be thought of as a more flexible and powerful version of a function pointer variable, allowing a user callback function to be called indirectly without knowing more than the interface to the function. Since the binder objects are templates, the type of the function and its arguments are not fixed but can be of any type. Also, member functions can be bound to an interrupt, which a callback function can never do.

The Binder system is powerful, yet in practice is quite simple to use. This system illustrates the power of the C++ language to contain a complicated system in a simple-to-use package.

Class InterruptHandler

This class is a base class for the ClassMemberHandler and FunctionHandler templates. It provides the interface the Pismo system uses to call the interrupt handler.

Class ClassMemberHandler Template

This template allows the binding of a member function of a class object with the object to call and an argument of any type. In this example the IsrHandler class is bound to a timer interrupt:

```
class IsrHandler
{
public:
    IsrHandler()
        : Binder(*this, &IsrHandler::MyHandler, &Tally), Tally(0)  { }
    ClassMemberHandler<IsrHandler, unsigned int *> Binder;

    void MyHandler(unsigned int * tally)
    {
        *tally += 1;

        if ((*tally & 0x7f) == 0)
            rtdx << "Isr tally: " << *tally << endl;
    }

private:
    // Data
    unsigned int    Tally;
};

// Instantiate a concrete instance of above class..
IsrHandler  Isr;

void IIMain()
{
    // Dynamically create an Irq object tripped from onchip timer 0
    Irq Timer0( intTimer0 );
    // Bind and install the interrupt vector
    Timer0.Install( Isr.Binder );
    // Program onchip timer 0 to signal at 100 Hz
    Timer0.Enable( false );
    DspClock Clock(100, 150, true, 0);
    Timer0.Enable( true );

    // Use RTDX event log to monitor progress
```

```
        rtdx.Enabled(true);

        rtdx << "Message from within IIMain,,,"<< endl;

        // Go to sleep...
        while (1)
            TSK_yield();
    }
```

In the above example, the handler uses a **int *** argument to pass out information from the interrupt routine.

Class FunctionHandler Template

This template allows the binding of stand-alone function with an argument of any type. In this example the OnTimerFired function is bound to a timer interrupt:

```
//
//  Timer Interrupt Handler Function
void  OnTimerFired(int arg);
//
//  Binder Object for Timer
typedef void (*IntFtnType)( int arg );

FunctionHandler<IntFtnType, int> TimerBinder(OnTimerFired, 0);
```

This is the installation of the handler in the program:

```
        //  Set up a real time clock to send commands to host on
        //    Target channel...
        Irq Timer0( intTimer0 );
        Timer0.Install( TimerBinder );
        Timer0.Enable( false );
        //
        //  Turn on the clock at 5 hz
        DspClock  Tclk0(50.0, 150.0);

        Timer0.Enable( true );
```

EDMA and QDMA Handling

The TI C6000 processor supports a rich, powerful DMA engine to move data without CPU intervention. There are two kinds of DMA allowed. One, EDMA is full featured but can take some time to set up. QDMA is TI's facility for quick DMA movement of data. It is similar to a normal DMA transfer except that it is software triggered and performs only a single transfer. No linking of blocks is permitted with QDMA. It also is faster to initiate as only a few registers need to be set to start a new transfer.

Both kinds of DMA use a set of registers to define the configuration of a DMA transfer. By properly configuring the settings, many different transfer types can be performed, such as interelaved data, two dimensional arrays, and so on. See the TI Peripheral Library guide for more information on configuring EDMA and QDMA.

The QDMA has a single set of configuration registers, so only one QDMA may be in progress at the same time. The EDMA has a pool of blocks that may be used to define simultaneous, complex transfers.

Class DmaSettings
The DmaSettings class manages an image of the settings registers used to configure a QDMA or EDMA transfer. It provides properties to read and set the individual fields of the registers, saving the user the effort of masking bits and shifting data. It even provides functions that preconfigure some commonly used transfers, saving even more programmer effort.

The following code fragment shows how the setter functions are used to set up for a transfer. The DmaSettings class returns a reference to self on all setter functions, allowing multiple parameters to be set on a single line:

```
DmaSettings Cfg;

Cfg.Priority(DmaSettings::priHigh).ElementSize(DmaSettings::is32bit)
Cfg.SourceIncr(DmaSettings::Incr).DestinationIncr(DmaSettings::Incr);
Cfg.TCInt(true).TCCode(1).FrameSync(true);
Cfg.SourceAddr((int)&src_array[0]).DestinationAddr((int)(dest_array+50));
Cfg.ElementCount(50).ElementIndex(1);
Cfg.FrameCount(0).FrameIndex(1);
```

Class Qdma
This class manages the posting of Qdma requests. It contains functions to allow configration of a transfer, initiating a transfer and completion notification via either an interrupt or a polling function. Because the system state is saved in the object, transfers can be predefined and saved to be posted at a later time.

As with all DMA objects, the Qdma object uses an internal DmaSettings object to define the transfer. The Settings() method provdes access to the object to allow calling the DmaSettings classes own configuration functions, or configurations can be loaded from a second object with the Load() method.

```
// Q is a Qdma object, here we change the destination address
Q.Settings().DestinationAddr((int)(dest_array+0x10));
```

For QDMA, a transfer is initiated when the parameters are loaded into the QDMA registers. This is performed by the `Submit()` method, which starts the preconfigured transaction, or loads the passed in configuration and submits it.

Only one Qdma transfer may be active in the system at one time. Multi-threaded applications must arbitrate Qdmas as appropriate.

If a terminal count interrupt is not used, a call for `WaitForComplete()` will delay until the completion occurs. `TestComplete()` will return a flag that can be used to check completion without blocking.

Qdma transfers may be configured to generate Terminal Count interrupts on completion of the transfer. Which TC bit is signalled is configured in the settings block.

A user supplied handler, similar to an interrupt handler, can be associated with the terminal count interrupt by a call to the `TcIntInstall()` method. The DMA system shares a single interrupt for all TC interrupts, and the system will call the installed handler when the particular bit in the TC register becomes set. The handler installer requires an Interrupt Binder Object (See “Interrupt Binder Templates” on page 85.) as an argument to associate a handler function or method and argument for the interrupt forwarding mechanism of Pismo. A second function, `TcIntDeinstall()` removes any installed handler.

Once installed, TC interrupts may be enabled or disabled by a call to `TcIntEnable()`.

The following example shows a full Qdma transfer with TC interrupt handling. In this example a class member function is bound to handle the interrupt response.

```
class DmaIsr
{
public:
    typedef void (*IntFtnType)(void * fallow);

    DmaIsr()
        : Binder(*this, &DmaIsr::MyHandler, NULL)  {  }

    void MyHandler(void * fallow)
    {
        qdma_not_done = false;
    }

    ClassMemberHandler<DmaIsr, void *> Binder;
};

DmaIsr Isr;

void IIMain()
{

    DmaSettings Cfg;
    Cfg.Priority(1).ElementSize(0).SourceIncr(1).DestinationIncr(1);
    Cfg.TCInt(true).TCCode(0);
    Cfg.SourceAddr((int)src_array).DestinationAddr((int)dest_array);
    Cfg.ElementCount(100).ElementIndex(1);
    Cfg.FrameCount(0).FrameIndex(1);
    Qdma Q(Cfg);

    // This QDMA operation will trip a terminal count interrupt when
    // all data has been moved.
    Q.TcIntInstall( Isr.Binder );
    InitArrays();
    Q.TcIntEnable(true);
    qdma_not_done = true;
    Q.Submit();
    while (qdma_not_done)
        ;
}
```

Class Edma

This class manages the posting of EDMA requests. It contains functions to allow configration of a transfer, initiating a transfer and completion notification via either an interrupt or a polling function. Because the system state is saved in the object, transfers can be predefined and saved to be posted at a later time.

An additional feature of EDMA is the ability to build complicated transfers by linking EDMA transfer blocks or by chaining EDMA transfers together.

For more information on EDMA, see the TI Peripheral Guide.

As with all DMA objects, the Edma object uses one or more internal DmaSettings object to define the transfer. One block is allocated for the primary transfer, and one for each linked block. The `Settings()` method provdes access to the primary transfer block's settings object. The `LinkSettings()` similarly allows to one of the link blocks's DmaSettings object. Each of these can be used to call DmaSetting's own configuration functions, or configurations can be loaded from a second object with the `Load()` method.

```
    // Ed is a Edma object, here we change the destination address
    Ed.Settings().DestinationAddr((int)(dest_array+0x10));
```

The EDMA transfer can be attached to one of a number of channels. To attach an EDMA to a hardware interrupt, use the channel with the same number as the hardware interrupt. For example, to attach an EDMA to external interrupt 4, use the EDMA channel 4.

For EDMA, before a transfer can be initiated, the parameters are loaded into the EDMA PRAM registers. This is performed by the `Submit()` method, which loads the PRAM with the transfer information. Unlike QDMA, this does not start the transfer itself. The transfer will be initiated when the associated hardware interrupt occurs. If using software triggering, use the `Set()` function to initiate a transfer. One `Set()` call is required for each link block in the transfer.

Each Edma transfer allocates blocks from the PRAM pool to configure its Link blocks. These blocks are a limited resource, and the alloction may fail. If the failure occurs, the `IsValid()` function will return false.

If a terminal count interrupt is not used, a call for `WaitForComplete()` will delay until the completion occurs. `TestComplete()` will return a flag that can be used to check completion without blocking.

Edma transfers may be configured to generate Terminal Count interrupts on completion of any and all blocks in the transfer. Which TC bit is signalled is configured in each settings block. This means there can be different handlers for different blocks in the transfer.

A user supplied handler, similar to an interrupt handler, can be associated with the terminal count interrupt by a call to the `TcIntInstall()` or `LinkTcIntInstall()` method. The Link function is used to install a handler for one of the link blocks as opposed to the primary block.

The DMA system shares a single interrupt for all TC interrupts, and the system will call the installed handler when the particular bit in the TC register becomes set. The handler installer requires an Interrupt Binder Object (See “Interrupt Binder Templates” on page 85.) as an argument to associate a handler function or method and argument for the interrupt forwarding mechanism of Pismo. A second pair of functions, `TcIntDeinstall()` and `LinkTcIntDeinstall()` removes any installed handler for the TC bit used by the block.

Once installed, TC interrupts for the entire transfer may be enabled or disabled by a call to `TcIntEnable()`.

The following example shows a full Edma transfer with TC interrupt handling. In this example a class member function is bound to handle the interrupt response.

```
class DmaIsr
{
public:
    typedef void (*IntFtnType)(void * fallow);

    DmaIsr()
        : Binder(*this, &DmaIsr::MyHandler, NULL)  {  }

    void MyHandler(void * fallow)
    {
        qdma_not_done = false;
    }

    ClassMemberHandler<DmaIsr, void *> Binder;
};

DmaIsr Isr;

void EdmaTest()
{
    Edma Ed;
```

```
    Ed.Settings().Priority(DmaSettings::priHigh).ElementSize(DmaSettings::is32bit);
    Ed.Settings().ElementIndex(1).ElementCount(50).FrameIndex(1).FrameCount(0);
    Ed.Settings().TCInt(true).TCCode(1).FrameSync(true);
    Ed.Settings().SourceAddr(int(&src_array[0])).SourceIncr(DmaSettings::Incr);
    Ed.Settings().DestinationAddr(dest_array).DestinationIncr(DmaSettings::Incr);
    //
    //  Define a linked
    DmaSettings Cfg;

    Cfg.Priority(1).ElementSize(0).SourceIncr(1).DestinationIncr(1);
    Cfg.TCInt(true).TCCode(1).FrameSync(true);
    Cfg.SourceAddr((int)&src_array[0]).DestinationAddr((int)(dest_array+50));
    Cfg.ElementCount(50).ElementIndex(1);
    Cfg.FrameCount(0).FrameIndex(1);

    Ed.AddLink(Cfg);
    Ed.LinkTcIntInstall( 0, Isr.Binder );
    Ed.TcIntClear();

    // This EDMA operation will trip a terminal count interrupt when
    // all data has been moved.
    InitArrays();
    Ed.TcIntEnable(true);
    qdma_not_done = true;
    Ed.Submit();
    // We software-initiate the EDMA here, but if this EDMA were using EINT4..7,
    // then an external int hardware pulse would remove need for Ed.Set, below
    Ed.Set();
    while (qdma_not_done)
        ;
    // Need to sync L2 cache with the of SDRAM, so that CPU can see the data
    CACHE_clean(CACHE_L2, dest_array, sizeof(dest_array));
    //
    //  Transfer the second transfer block...
    Ed.Set();
    while (qdma_not_done)
        ;
    // Need to sync L2 cache with the of SDRAM, so that CPU can see the data
    CACHE_clean(CACHE_L2, dest_array, sizeof(dest_array));
}
```

The above example sets up a two block linked transfer triggered by software. A TC Interrupt is configured to signal the completion of each block in the transfer. The mainline waits for each block transfer to finish, as notified by the interrupt handler. Then the next block transfer is triggered by a second call to `Set()`. The Cache functions are required to assure that the cache and memory contents are back in synchronization.

Linked and Chained blocks

EDMA transfers may span multiple transfer blocks. On the completion of the primary transfer, the first link block is loaded into the primary block and initiated. When this block completes, the next linked block is loaded, and so on. A link block can form a loop, but it is important to remember that the primary block can never be part of a loop. Since it is overwritten by the first linked transfer, this transfer can only occur once. Because of this to make a loop of two transfers requires **three** blocks to be configured. The primary block contains the first transfer, the first link the second transfer, and the third is a repeat of the first transfer that is linked back to the first link block.

Link blocks are allocated by a call to `AddLink()`. This call automatically configures the preceding block to link to this newly added block. It returns the index of the newly added block that can be used in order to configure the link block. To form a closed loop in a block chain, call `LinkBackTo()`. This connects the final block in the chain back to the block whose index is given in the argument.

Transfer chaining is a mechanism for having a transfer trigger another on completion. The `ChainTo()` and `ChainEnable()` methods set up a chaining relation between two transfers. Note that on the TI C671x processor, the second transfer must be configured on channels 8-11.

Class EdmaMaster
This class acts as a holder for functions and information common to all EDMA interrupts instead of associated with a single EDMA channel. Only one instance of EdmaMaster is created at program initialization. It is accessed by calling the static member function `EdmaMaster::Object()`.

EdmaMaster contains several functions dealing with the EDMA PRAM. This is a memory region shared among all EDMA objects giving a common storage for configuration blocks. This is a limited resource, so be wary of allocating many Edma blocks and not releasing them. The method `ClearPram()` clears all the PRAM blocks in a single operation.

EdmaMaster contains several functions dealing with the EDMA PRAM. This is a memory region shared among all EDMA objects giving a common storage for configuration blocks. This is a limited resource, so be wary of allocating many Edma blocks and not releasing them. Also available are functions to give access to the area at the end of the PRAM that is not used by the system. This scratchpad memory might be of use as a shared memory pool in an application.

Quixote Example Programs

Under **Quixote\Examples** in the install directory, the baseboard's example programs are installed. Some examples have no host component, and some use the terminal emulator applet as the host. Host examples are written in C++ either under Borland C++ Builder or Microsoft MSVC, or both. Target examples are written using CCS 3.x and DSP/BIOS.

Table 16. Quixote Example Programs

Example	Host	Target	Illustrates
AEcho	terminal emulator	DSP/BIOS	Use of DSP/BIOS drivers; Analog output driven from analog input
AnalogIn	terminal emulator	DSP/BIOS	Analog capture into DSP memory. Samples sent to host and saved for analysis.
AnalogOut	terminal emulator	DSP/BIOS	Analog waveform playback from pre-calculated buffer in DSP memory.
ASnap	BCB, VC++7, VC8 (.NET)	DSP/BIOS	Full-rate analog capture to memory, then send to host via busmastering. Illustrates data capture and target/host communication with commands and data packets.
CommonCDB	N/A	N/A	Shared CDB file for all examples.
CpuInRate	VC++7, VC8 (.NET)	DSP/BIOS	Shows data transfer from host machine in to the target CPU.

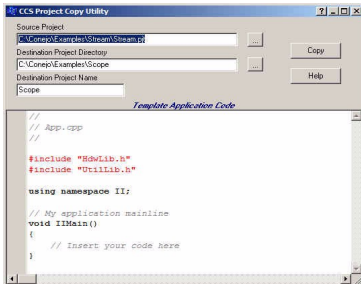
Example	Host	Target	Illustrates
CpuOutRate	VC++7, VC8 (.NET)	DSP/BIOS	Shows data transfer from the target CPU out to the host machine.
Edma	terminal emulator	DSP/BIOS	Use of Pismo Edma and Qdma wrapper classes with installable interrupt handlers.
FftFix	terminal emulator	DSP/BIOS	Use of Fourier class to perform foward and inverse FFTs
Files	terminal emulator	DSP/BIOS	Use of C++ Standard I/O library
FirFix	terminal emulator	DSP/BIOS	Use of BlockFir class to perform FIR filter functions.
FramedStimulusResponse	terminal emulator	DSP/BIOS	Output a stimulus signal and read in response on the input.
Swi	terminal emulator	DSP/BIOS	Use of Pismo SoftInt class for software interrupts.
Timer	terminal emulator	DSP/BIOS	Use of Pismo ClockBase objects for timebase control.
UserData	terminal emulator	DSP/BIOS	Use of baseboard User digital I/O

The Next Step: Developing Custom Code

In building custom code for an application, Innovative Integration recommends that you begin with one of the sample programs as an example and extend it to serve the exact needs of the particular job, or at least refer to the examples to see how some functions are done. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

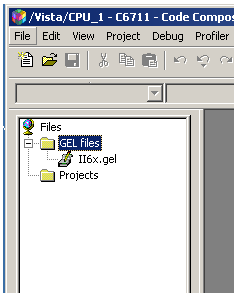
Building a Target DSP Project

Building a project suitable for a Matador or Velocia baseboard requires a particular setup of the project.

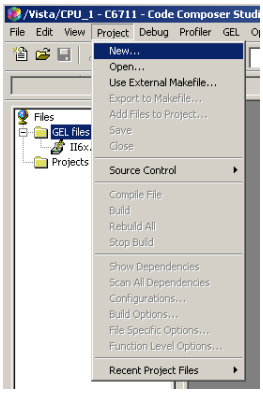


By far, the easiest way to create a new DSP project is by using an existing project as a template. The CopyCcsProject applet provided in the Pismo Toolset automates this task. To use this utility, select an existing Code Composer project as the Source Project, typically one of the example programs supplied in the Pismo Toolset. Next, select the directory into which you wish the new project to be created, using the Destination Project Directory edit control. Then, edit the Destination Project Name for the newly-created project. Finally, click the Copy button to create the new project from the template. The new project may be opened and used within Code Composer.

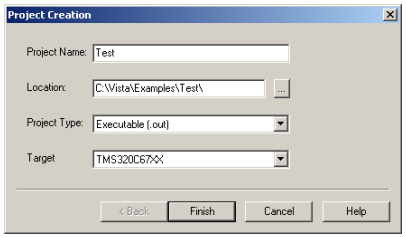
Alternately, you may follow the manual steps below to create a new target DSP project. The project name used below is called Test, but you should name your project appropriately for your application.



Start Code Composer Studio. In the default configuration, the project window will contain no projects but will contain the default Innovative-supplied board initialization GEL file.

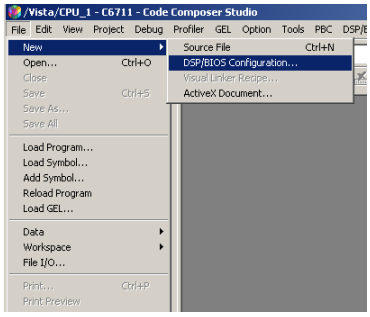


Click Project | New on the menu bar to create a new DSP project.

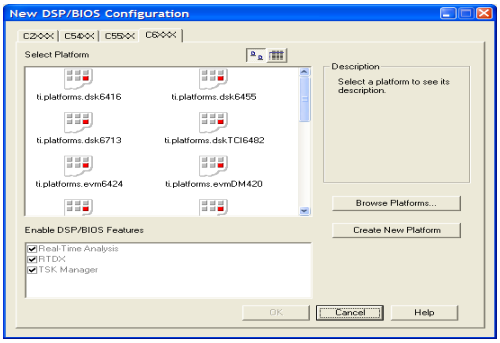


Specify the location for the new project and its name. In this example, a new project called Test is being created in the `Sbc6711-Pismo\\Examples\\` directory. Change the location to accommodate your board type and processor type.

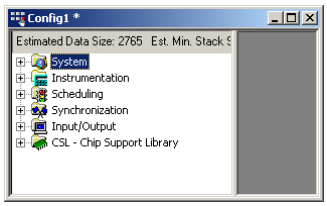
After the new project has been created, it will appear in the CCS project window under the Projects folder.



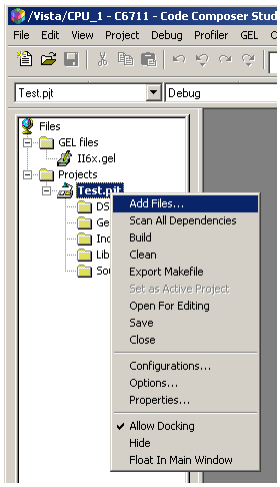
Click File | New | DSP/BIOS Configuration to create a new TCF file for use in the project.



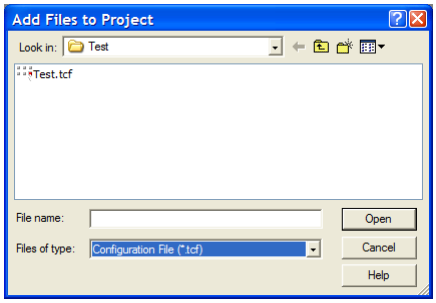
Select the relevant template for the baseboard from the list in the New DSP/BIOS Configuration dialog box.



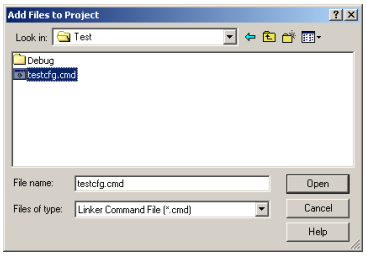
By default, this TCF will be named Configuration1. Save it as `Test.TCF`.



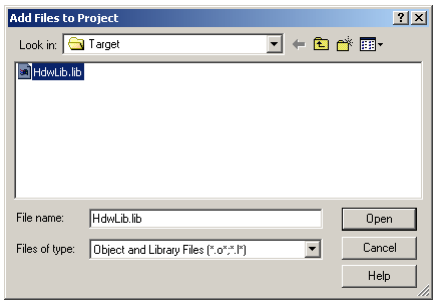
Though the TCF and its support files have been created on disk, you must manually add them to the Test project. Right-click on `Test.pjt` in the Project window to invoke the project hot menu. Click Add Files to add a file to the project.



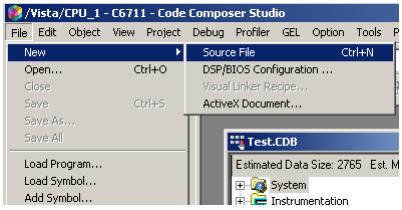
Select the the newly-created `Test.tcf` for addition to the project. This will implicitly add the auto-generated files `Testcfg.s62 (Testcfg.s64 for Velocia cards)` and `Testcfg_c.c` to the project as well.



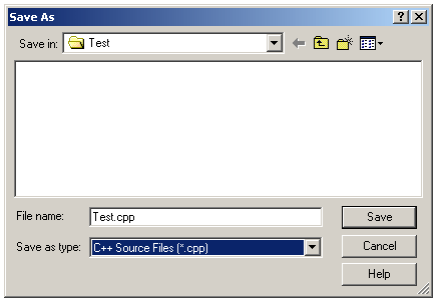
Right-click on `Test.pjt` in the project window, click “Add Files” then select the the newly-created `Test.cmd` for addition to the project.



Right-click on `Test.pjt` in the project window, select Add Files, then browse to the Examples directory and select `Examples.cmd` for addition to the project.



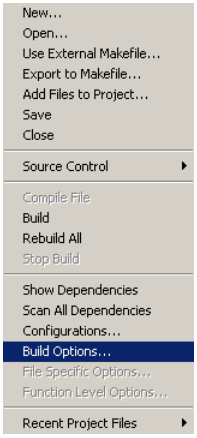
Add an new C++ source file to the project. Click File | New | Source File to create an empty source document.

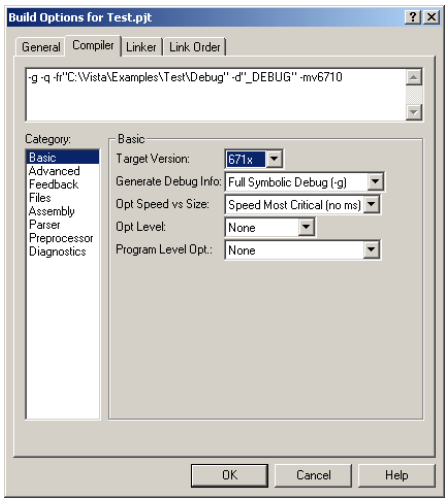


Rename the new source document to `Test.cpp`. To use the Pismo libraries, you must use C++ files and the C++ compiler, even if you intend to restrict your own coding to the C subset of C++

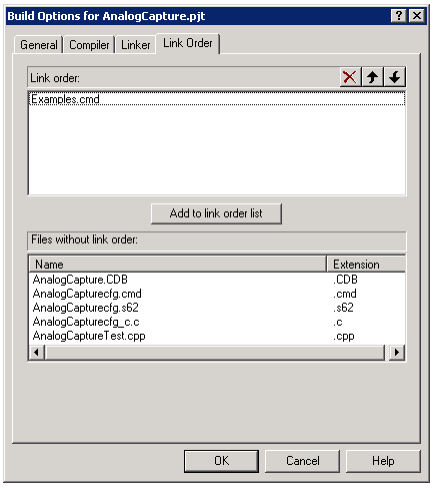
```
//
// App.cpp
//

#include "Pismo.h"

using namespace II;

// Application mainline
void IIMain()
{
    // Insert your code here
}
```

Type the boilerplate code below into your source file. This is the minimum code needed for any Pismo C++ application.

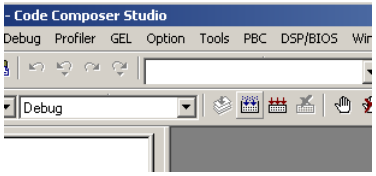




Click the menu Project | Build Options to invoke the compiler Build Options dialog. Then select the Files Category, then enter the pathspec to the `Examples.opt` file in the Examples directory to the Options File edit box.



Click on the Link Order tab, then add `Examples.cmd` to the Link Order List.



Click the Incremental Build button to rebuild the template application. It should compile and link without errors.

Writing a Program

The basic program given in the example above includes a 'Main' function `IIMain()`. DSP/BIOS, the OS used in the Pismo library, uses code inserted after exiting from the normal C-language `main()` to initialize features of DSP/BIOS. This means that some language features are not available then. To avoid these problems the Pismo library provides a `main()` function and uses it to create a single thread. This thread, when executed, calls the `IIMain()` function. Inside of this thread, all DSP/BIOS is initialized and ready for use. It is required that the user include this function, and use it as the equivalent of the old main process function in C.

Host Tools for Target Application Development

The Innovative Integration Pismo Toolset allows users of Innovative DSP processor boards to develop complete executable applications suitable for use on the target platform. The environment suite consists of the TI Optimizing C++ Compiler, Assembler and Linker, the Code Composer debugger and code authoring environment as well as Innovative's custom Windows applets (such as the terminal emulator).

Code Composer Studio is the package used to automate executable build operations within Innovative's Pismo Toolsets, simplifying the edit-compile-test cycle. Source is edited, compiled, and built within Code Composer Studio, then downloaded to the target and tested within either the Code Composer Studio debugger or via the terminal emulator.

Code Composer Studio may be used for both code authoring and code debugging. Details of constructing projects for use on Innovative DSP platforms are given in the above section of this chapter.

Do not confuse the creation of target applications (code running on the target DSP processor) with the creation of host applications (code running on the host platform). The TI tools generate code for the TI DSP processors, and are a separate toolset from that needed to create applications for the host platform (which would consist of some native compiler for the host processor, such as Microsoft's Visual C++ or Borland Builder C++ for IBM compatibles). To create a completely turnkey application with custom target and host software, *two* programs must be written for two separate compilers. While Innovative supports the use of Microsoft C/C++ for generation of host applications under Windows with sample applications and libraries, we do not supply the host tools as part of the Development Environment. For more information on creating host applications, see the section in this manual on host code development.

This section supplies information on the use of the development environment in creating custom or semicustom target DSP software. It is not intended as a primer on the C++ language. For information on C/C++ language basics, consult one of the primer books available at your local bookstore.

Components of Target Code (.cpp, .tcf, .cmd, .pjt)

In general, DSP applications written in TI C++ require at least three files: a .cpp file (or "source" file) containing the C++ source code for the application a .cmd file (or "command" file) which contains the target-specific memory-map and build data needed by the linker, a .tcf file (or "command database" file) which specifies the properties of the BIOS operating system used within the application and a .pjt file ("project" file) which centralizes all project-specific options, settings and files. There may also be one or more .asm assembler source files, if the user has coded any portions of the application in assembly language.

Edit-Compile-Test Cycle using Code Composer Studio

Nearly every computer programming effort can be broken down into a three step cycle commonly known as the edit-compile-test cycle. Each iteration of the cycle involves editing the source (either to create the original code or modify existing code), followed by compiling (which compiles the source and creates, or builds, the executable object file), and finally downloading and testing the result to see if it functions in the desired fashion. In the Innovative Integration development system these stages are accomplished within the Code Composer integrated development environment (IDE).

By using Code Composer Studio, these stages of the programming cycle are accomplished entirely within the IDE. The project features of Code Composer Studio support component file editing and compilation stages, along with allowing the executable result to be downloaded and tested on the target hardware. This fully integrated programmers environment is more user-friendly then the basic command line interface, which comes standard with the TI tools.

Automatic projectfile creation

When a project is created, opened, modified, built or rebuilt, the Code Composer Studio dependency generator automatically generates a project makefile (named `<project file>.pjt`, located in the project directory), which is capable of rebuilding the project's output file from its components.

This file is automatically submitted to the internal make facility whenever you click on build or rebuild within Code Composer Studio. The make facility automatically constructs the output file by recompiling the out-of-date source files including the dependencies contained within those source files.

Rebuilding a Project

It is sometimes necessary to force a complete rebuild of an output file manually, such as when you change optimization levels within a project. To force a project rebuild, select Project | Rebuild All from the Code Composer Studio menu bar.

IIMain replaces main.
Due to restrictions within Dsp/Bios, not all BIOS features may be safely used within `main()`, since it is called early in the system initialization sequence. To circumvent this limitation, Pismo automatically constructs a default thread running within normal priority and starts this thread automatically. The entry point function in this thread is called IIMain, and all Pismo applications must define this function. This function is intended to replace main in your application programs. You may safely call any BIOS function within IIMain.

Running the Target Executable

The test program may be converted into a simple, "Hello World!" example, by using the built-in standard I/O features within Pismo. Bring up the `Test.cpp` source file edit screen. Scroll down the source file by using cursor down button until you reach the `IIMain()` function. Edit it as follows:

```
#include "HdwLib.h"
#include "UtilLib.h"

        cio << init;
cio << "Hello World!" << endl;
```

```
cio.monitor();
```

You can now compile the new version by executing Build from the Project menu (or by clicking on its toolbar icon). This causes Code Composer Studio to start the compiler, which produces an assembly language output. The compiler then automatically starts the assembler, which produces a `.obj` output file (`test.obj`). Code Composer Studio then invokes the TI Linker using the `testcfg.cmd` file, which is located in the project directory. This rebuilds the executable file using the newly revised `test.obj` . If no errors were encountered, this process creates the downloadable COFF file `test.out`, which can be run on the target board. At this point, the program may be run using the terminal emulator applet, which may be invoked using the terminal emulator shortcut located within the target board program group created during the Pismo Libraries installation process. In the terminal emulator, download the `test.out` file. The program runs and outputs the message "Hello, World" to the terminal emulator window.

If errors are encountered in the process, Code Composer Studio detects them and places them in the build output window. If the error occurred in the compiler or assembler (such as a C++ syntax error), the cursor may be moved to the offending line by simply double-clicking on the error line within the build output window, and the error message will be displayed in the Code Composer Studio status bar. If the linker returns a build error, the build output window shows the error file. From this information, the linker failure can be determined and corrected. For example, if a function name in a call is misspelled, the linker will fail to resolve the reference during link time and will error out. This error will be displayed on the screen in the build output window.

Note:
Be sure to start the terminal emulator **BEFORE** starting Code Composer, to avoid resetting the DSP target in the midst of the debugging session. If the terminal emulator is not yet running and you wish to run the Test object file, perform the following steps.

1. Execute Debug | Run Free to logically disconnect the DSP from the debugger software.

2. Terminate the Code Composer Studio application.

3. Invoke the terminal emulator application.

4. Restart the Code Composer Studio application.

This outlines the basics of how to recompile the existing sample programs within the Code Composer Studio environment.

Anatomy of a Target Program

While not providing much in the way of functionality, the `test` program does demonstrate the code sequence necessary to properly initialization the target. The exact coding, however, is very specific to the I.I. C Development Environment, target boards, and is explained in this section in order to acquaint developers with the basic syntax of a typical application program.

```
/*
*       HELLO.CPP
*       Test file/program for target board.
*/

#include "Pismo.h"
```

```
IIMain()
{
        cio << init;

        cio << "Hello World!" << endl;
        cio << "\nEchoing keystrokes..." << endl;

        char key;

        do
                {
                cio >> key;
                cio << key << flush;
                }
        while(key != 0x1b);

        cio.monitor();
}
```

The two lines of the program that being with a "#" are #include statements, which include the header files for the hardware and utility I/O libraries. These include prototypes for all the library classes within Pismo.

The `cio << init` invocation will setup the standard monitor I/O interface and reset the terminal window. The next lines perform the basic standard I/O functions of printing "Hello World!" & "Echoing keystrokes...". These two lines are where custom code could be inserted.

The following do-loop sequence simply echoes keys typed at the terminal emulator back to the terminal display, until the `Esc` key is pressed. When `Esc` is pressed, the `cio.monitor()` function effectively terminates the program, except that interrupts are still active and interrupt handlers (if they had been installed) would still execute properly.

The `test` program is very simple, but it contains the basic components of a typical DSP application, as well as the initialization needed to interact with the terminal emulator.

Use of Library Code

Library routines can be compiled and linked into your custom software simply by making the appropriate call in the source and adding the appropriate library to the linker command file. Refer to the library reference within the Pismo online help for library location information on each class and method.

In general, user software needs to #include the relevant library header file in source code. The header files define prototypes for all library functions as well as definitions for various data structures used by the library functions. The files `HdwLib.h` and `UtilLib.h` should be included within all programs; The file `DspLib.h` should be included if a program uses functions in the DspLib signal processing library.

Example Programs

Under **<baseboard>\Examples** in the install directory, the baseboard's example programs are installed. Some examples have no host component, and some use the terminal emulator applet as the host. Host examples are written in C++ either under

Borland Builder or Microsoft MSVC, or both. Target examples are written using CCS 3.3 and DSP/BIOS. Note that not all of the examples listed below are available for all targets.

Table 17. Pismo Example Programs

Example	Host	Target	Illustrates
FftFix	terminal emulator	DSP/BIOS	Use of Fourier class to perform forward and inverse FFTs
FirFix	terminal emulator	DSP/BIOS	Use of BlockFir class to perform FIR filter functions.
Edma	terminal emulator	DSP/BIOS	Use of Pismo Edma and Qdma wrapper classes with installable interrupt handlers.
Files	terminal emulator	DSP/BIOS	Use of C++ Standard I/O library
CpuInRate	BCB MSVC	DSP/BIOS	Use of Target to Host message and data packet passing via PCI bus.
CpuOutRate	BCB MSVC	DSP/BIOS	Use of Host to Target message and data packet passing via PCI bus.
LinkPort	BCB	DSP/BIOS	Use of LinkPort driver to flow data between all processor in mesh
Swi	terminal emulator	DSP/BIOS	Use of Pismo SoftInt class for software interrupts.
Timer	terminal emulator	DSP/BIOS	Use of Pismo ClockBase objects for timebase control.

The Next Step: Developing Custom Code

In building custom code for an application, Innovative Integration recommends that you begin with one of the sample programs as an example and extend it to serve the exact needs of the particular job. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

Writing Custom Applications

Most scientific and engineering applications require the acquisition and storage of data for analysis after the fact. Even in cases where most data analysis is done in place, there is usually a requirement that some data be saved to monitor the system. In many cases a pure data logger that does no immediate processing is the most common application.

Because of the high data rate of the Quixote baseboard, a logger that saves all of the data to the host disk is impossible. Instead, the best that can be done is to capture a relatively large set of samples out of the data stream, and send that to the host.

The Asnap Example

The Asnap example in the Quixote software distribution demonstrates this kind of capture application. It consists of a host program in Windows that controls and communicates with a target program on the Quixote's DSP. Commands and Data move between the two programs, using Innovative's software libraries to accomplish the task.

Tools Required

In general, writing applications for the Quixote requires the simultaneous development of *two* programs – one for the host and one for the target. Each of these requires a development environment, a debugger, and a set of support libraries from Innovative.

Table 18. Development Tools for the Asnap Example

Processor	Development Environment	Innovative Toolset	Project Directory
Target DSP	Code Composer Studio 3.0	Pismo	Examples\ASnap
Host PC	Borland C++ Builder 6.0	Malibu	Examples\ASnap\Bcb
	Microsoft Visual Studio 2003		Examples\ASnap\VC7
	Microsoft Visual Studio 2005		Examples\ASnap\VC8
	Common Host Code		Examples\ASnap\Common

On the host side, the Malibu library is source code compatible with all three of the above environments. The code that performs much of the actual functioning of the program, outside of the User Interface portion of the program, is therefore common code. Each project uses the same file to interact with the hardware and acquire data.

Program Design

The Asnap example is designed to allow repeated data taking runs on command from the host. Because of the high data rate of the analog hardware, we know we can not take data forever without falling behind. However we maximize our chances by setting up the system to capture blocks into target memory as fast as possible until a specific amount is read in. Then the data taking is stopped and the accumulated data delivered to the host and logged.

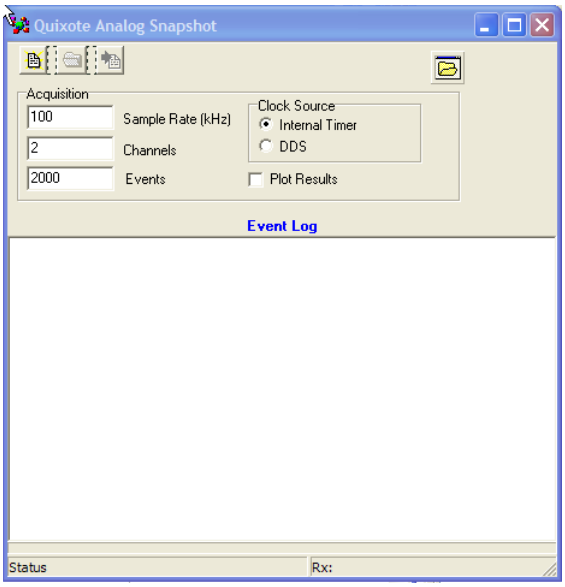
The example uses the Messages to send commands and parameters to the target to control the I/O. Busmaster block transfer is used to deliver data to the host.

After delivering the data the target resets itself for further commands from the host so that repeated data snapshots can be taken.

The Host Application

The picture to the left shows the main window of the Asnap example. This form is from the designer of the Borland C++ Builder version of the example. It shows the layout of the controls of the User Interface.

User Interface



On the top of the form is a button bar than contains control buttons for the example. These buttons are how the user initiates an action in the example.

From left to right, the buttons perform these actions:

- Logic Loading. The user logic of the Quixote must be loaded at least once per session (it remains valid until power is removed from the board). This button performs the logic loading from an EXO File.
- COFF Loading. Similarly, the DSP must be loaded with a target program COFF File. This button controls this.
- Capture. This button controls a single capture of data following the parameters given in the UI.

The square to the left is a non-visual component in Builder that controls the posting of a File Open Dialog box. It will not appear in the running application.

Below that is a set of controls that hold the parameters of the acquisition. These settings are delivered to the target and configure the target program accordingly.

The Event Log, Progress Bar and Status Bar at the bottom display progress messages and feedback during the operation of the program.

Host Side Program Organization

The Malibu library is designed to be rebuildable in each of three different host environments: Borland C++ Builder, Microsoft Visual Studio 2003, and Microsoft Visual Studio 2005 using the .NET UI. Because the library has a common interface in all environments, the code that interacts with Malibu is separated out into a class, ApplicationIo in the files ApplicationIo.cpp and .h. This class acts identically in all the platforms.

The Main form of the application creates an *ApplicationIo* to perform the work of the example. The UI can call the methods of the *ApplicationIo* to perform the work when, for example, a button is pressed or a control changed.

Sometimes, however, the *ApplicationIo* object needs to 'call back into' the UI. But since the code here is common, it can't use a pointer to the main window or form, as this would make *ApplicationIo* have to know details of Borland or the VC environment in use.

The standard solution to decouple the *ApplicationIo* from the form is to use an Interface class to hide the implementation. An interface class is an abstract class that defines a set of methods that can be called by a client class (here, *ApplicationIo*). The other class produces an implementation of the Interface by either multiple inheriting from the interface, or by creating a separate helper class object that derives from the interface. In either case the implementing class forwards the call to the UI form class to perform the action. *ApplicationIo* only has to know how to deal with a pointer to a class that implements the interface, and all UI dependencies are hidden.

The predefined *IUserInterface* interface class is defined in ApplicationIo.h. The constructor of *ApplicationIo* requires a pointer to the interface, which is saved and used to perform the actual updates to the UI inside of *ApplicationIo's* methods.

ApplicationIo

Initialization

The main form creates an ApplicationIo object in its constructor. The object creates a number of Malibu objects at once as can be seen from this detail from the header ApplicationIo.h.

```
private:
    //
    //  Member Data
    IUserInterface *              UI;
    Innovative::C64xDsp           Dsp;
    Innovative::Quixote           Board;
    Innovative::TiBusmasterStream Stream;
    Innovative::DataLogger        Logger;
    Innovative::BinView           BinView;
```

In Malibu, objects are defined to represent units of hardware as well as software units. The C64xDsp and Quixote objects represent the baseboard and the CPU on it. The TiBusmasterStream object encapsulates the I/O operations supported by the CPU. The Logger and BinView objects support logging data to disk and the BinView data analysis application.

Creating a hardware object such as Quixote does not attach it to the hardware. The object has to be explicitly opened. The OpenBoards() method sets up the hardware objects:

```
//-------------------------------------------------------------------------
//  ApplicationIo::OpenBoards() -- Open Hardware & set up callbacks
//-------------------------------------------------------------------------
```

```
void  ApplicationIo::OpenBoards(int dspTarget)
{
    Board.Target(dspTarget);
    Board.Open();
    Board.Reset();

    Board.Logic().OnFpgaParseProgress.SetEvent(this, &ApplicationIo::HandleParseProgress);
    Board.Logic().OnFpgaParseComplete.SetEvent(this, &ApplicationIo::HandleParseComplete);
    Board.Logic().OnFpgaParseMessage.SetEvent(this, &ApplicationIo::HandleLoadError);
    Board.Logic().OnFpgaLoadProgress.SetEvent(this, &ApplicationIo::HandleLoadProgress);
    Board.Logic().OnFpgaLoadComplete.SetEvent(this, &ApplicationIo::HandleLoadComplete);
    Board.Logic().OnFpgaLoadMessage.SetEvent(this, &ApplicationIo::HandleLoadError);
```

This code configures the Quixote object. First the identifier target number is set, then the Open() method attaches the object to the actual physical device. The board is then reset.

The next lines configure the Quixote logic loader's informational events. Malibu has a method where functions can be 'plugged into' the library to be called at certain times or in response to certain events detected. Events allow a tight integration between an application and the library. These events are informational messages called by the logic loader feature of the Quixote. They display feedback during the loading of the user logic.

```
    Dsp.Target(dspTarget);
    Dsp.Open();
    UI->AppendToLog("Quixote Opened!");
    Dsp.Reset();
```

This code initializes the DSP on the Quixote. Note the use of the UI interface class object (UI) to perform a UI function. The AppendToLog() method adds the text to the log display on the main window.

```
    Stream.ConnectTo(&Dsp);
    Stream.OnPacketAvailable.SetEvent(this, &ApplicationIo::PacketAvailableHandler);
    Stream.OnPacketAvailable.Synchronize();
    Stream.OnMailAvailable.SetEvent(this, &ApplicationIo::MailAvailableHandler);
    Stream.OnMailAvailable.Synchronize();
```

The Stream object is an 'stream' object that manages communication between the application and a piece of hardware. Separating the I/O into a separate class clarifies the distinction between an I/O protocol and the implementing hardware. Different types of hardware can all support a single type of streaming. Also, it is possible that a single type of hardware could support several styles of I/O, each with its own distinct interface.

A I/O Stream class is attached to a piece of hardware via the ConnectTo() method. If an objects supports a stream type, this call will be implemented. Unsupported stream types will not compile. We then attach handlers to important events.

The TiBusmasterStream supports two types of communication: a 'Mail' message transfer that sends small packets and a busmastered Packet transfer for large data transfers. An event is defined that will be called when each of these types of data arrives from the target, and we intercept both of them here.

An event is not necessarily called in the same thread as the UI. If it is not, and you want to call a UI function in the handler you have to have the event synchronized with the UI thread. The call to Synchronize() directs the event to call the event handler in the main UI thread context. This results in a slight performance penalty, but allows us to call UI methods in the event handler freely.

```
    //
    //  Hook progress message
    Dsp.Cpu().OnCoffLoadProgress.SetEvent(this, &ApplicationIo::DownloadProgress);

    Dsp.Cpu().Boot();
```

```
}
```

The final line boots the CPU so that the debugger can be started.

Logic Loading

Because the logic must be loaded for Quixote applications to function, the example includes a logic loader button. Let's follow the button press code through the entire operation.

In the UI, when the logic load button is pressed a handler method is called. Here it is attached to the method LogicLoadBtnClick():

```
//---------------------------------------------------------------------------
//  TForm1::LogicLoadBtnClick() -- Load Logic
//---------------------------------------------------------------------------

void __fastcall TForm1::LogicLoadBtnClick(TObject *Sender)
{
    OpenDialog->Title = "Logic File Selection";
    OpenDialog->Filter = "Logic Files (*.exo)|*.EXO|All Files (*.*)|*.*";

    OpenDialog->FileName = "";
    OpenDialog->InitialDir = ExtractFilePath(AppPath.c_str());

    OpenDialog->Options.Clear();
    OpenDialog->Options << ofFileMustExist << ofHideReadOnly << ofNoChangeDir;

    if (!OpenDialog->Execute())
        return;
```

The code above posts a dialog allowing the user to select a logic file. If the user cancels out, no logic is loaded. If he selects a file, we move on to the loading.

```
    // don't let this be clicked again until download completes
    ProgressBar->Position = 0;
    ProgressBar->Max = 100;
    LogicLoadBtn->Enabled = false;

    Io->LoadLogic(std::string(OpenDialog->FileName.c_str()));

}
```

We do some more UI tricks, such as setting up the progress bar limits and disabling the load button, before we get to the meat of the handler. We extract the file name from the dialog, and pass it to the ApplicationIo method LoadLogic().

```
//---------------------------------------------------------------------------
//  ApplicationIo::LoadLogic() --
//---------------------------------------------------------------------------

void  ApplicationIo::LoadLogic(const std::string & Image)
{
    Board.Reset();

    UI->AppendToLog("------------------------------------------------");
    UI->AppendToLog("  Parsing logic file");

    UI->UpdateLogicLoadProgress(0);
```

The handler resets the board, then uses the UI callback interface to do more setting up of the UI for the logic loading. The interface allows UI calls to be made here in the Io object or out in the UI class itself, whichever is more convenient.

```
    Board.Logic().ConfigureFpga(Image);
}
```

And here we have a call into Malibu. This starts the long sequence of logic loading. The call returns at once, as the loading is managed by a background thread. Status and feedback are given by events that can be intercepted by the application or not.

```
//-----------------------------------------------------------------------------
//  ApplicationIo::HandleLoadProgress() --
//-----------------------------------------------------------------------------

void ApplicationIo::HandleLoadProgress(Innovative::ProcessProgressEvent & event)
{
    UI->UpdateLogicLoadProgress( event.Percent );
}
```

This event is called to give a percentage progress of the entire operation. The handler calls back to the UI, where a Progress bar control is updated to give a visible indication of the loading progress. Knowing the events to handle is a large part of operating with the Malibu library.

```
//-----------------------------------------------------------------------------
//  ApplicationIo::HandleLoadComplete() --
//-----------------------------------------------------------------------------

void ApplicationIo::HandleLoadComplete(Innovative::ProcessCompletionEvent & event)
{
    UI->UpdateLogicLoadProgress( 100 );
    UI->AppendToLog("Load completed ok");
    UI->OnLogicLoadComplete();
}
```

Finally, the logic loader sends this event when the load is finally done. In this case, all we do is update the UI so the user can see that the load is finished. In other cases this could trigger the application to automatically perform additional actions.

COFF File Loading

The next stage of the configuration is to load code onto the target CPU. Again, this is triggered by a button on the UI, which posts a dialog allowing a file to be selected. Then it calls this method of the ApplicationIo object:

```
//-----------------------------------------------------------------------------
//  ApplicationIo::DownloadCoff() -- Coff Loading
//-----------------------------------------------------------------------------

void  ApplicationIo::DownloadCoff(const std::string & filename)
{
    Dsp.Cpu().DownloadCoff(filename);
}
```

Again, a single method performs the entire COFF load – or at least *begins* the COFF loading. Like logic loading, this call initiates a COFF load that is actually performed in the background. You can use events to get feedback about the progress of the load. After the COFF load is complete, the target program is started.

In this case, however, even knowing when the load is complete is not enough. The target program has to set up itself for communication and the host must wait until this is complete before sending commands or data to the target. This is the point where the two programs have to begin cooperating. The solution we use is to have all target programs send a “Login”

message to the host at the end of their initialization. This message signals to the host that the target is ready for communications to begin. This snippet from the application message handler shows this login message response:

```
            case ccLogin:
                UI->AppendToLog("ccLogin");
                UI->AppendToLog("Dsp logged in: " + IntToString(++LoginTally));
                UI->OnLoginCommand();
                break;
```

This prints message to the log and activates the 'Snap' button that begins data transfers.

Starting a Data Capture

After downloading, the Snap button is activated to enable capturing data.

```
//---------------------------------------------------------------------------
//  ApplicationIo::BeginSnap() -- Start data taking
//---------------------------------------------------------------------------
//  This function is executed when the user clicks on the RcvButton.
//  It causes the ccDataXferMsg command to be sent to the target DSP,
//  which responds by sending a buffer of contiguous analog acquisition data

void  ApplicationIo::BeginSnap(const std::string & data_file_base)
{
    RcvMessageTally = RcvByteTally = 0;
    //
    //  Logger saves the data transmitted into a file if checking data
    Logger.Stop();

    Logger.FileName = AppPath + data_file_base + ".bin";
    Logger.Start();

    UI->GetSettings();
```

This UI interface command loads the settings information from the UI controls into the Settings structure in the ApplicationIo class. This class is declared as publicly accessible to reduce the amount of access methods in the class.

```
    MatadorMessage Msg;
    Msg.TypeCode(ccDataXferMsg);
    Msg.AsFloat(0, Settings.SampleRate);
    Msg.Data(1, Settings.Events);
    Msg.Data(2, Settings.Channels );
    Msg.Data(3, Settings.ClockSource );
    Msg.Data(4, Settings.Plot );

    //
    //  Send the command to send data buffers on target to host
    Stream.Send(Msg);
```

The configuration information is loaded into a MatadorMessage object for transmission to the target. The data is a 16 word packet, the first two of which are reserved for header information. We use the TypeCode() field to define what type of message this is. Data can be loaded as different data types. The Sample Rate is transferred as floating point, while the others are transferred as integers.

The Stream's Send() method sends the packet to the target as a mail command packet.

```
    UI->AppendToLog("-------------------------------------------------------------");
    UI->AppendToLog("Capturing...");
}
```

At this point, the action shifts to the target side program. The next section will discuss the target code.

The Target Application

Main Thread

On the target we use the Pismo libraries to access the hardware. These libraries are built on the TI DSP/BIOS operating system that provides a multithreaded working environment. The library launches a starting thread that calls the IIMain() function. This thread starts first, acting much like the main() function in a standard C program.

```
void IIMain()
{
        CaptureThread Capture(tpHigher);
        Capture.Resume();
```

The data acquisition requires its own thread, which is created and started here. Its priority is elevated to assure that data taking gets executed first when it needs servicing.

```
        MessageTransfer Xfr;
        IIMessage C;

        for (;;)
            {
            //
            //  Wait for message to arrive from host
            Xfr.Recv(0, C);
            DispatchPacket(Capture, C);
            }
```

We also need a thread dedicated to responding to messages from the host, so we use this one. The Recv() method waits for a message packet to arrive from the host. The DispatchPacket() function then processes the message. When we left the host, we had just issued a command packet to the target, thus waking up this thread to process it.

```
}
```

The Capture Thread

The capture thread is a class derived from the Pismo Thread class. It manages the creation and use of threaded code. To use it you derive your class from thread:

```
class CaptureThread : public Thread
{
....
```

The Thread class creates a thread that executes the Execute() method in a separated thread. You therefore have to override that method to perform the work of the thread:

```
        void Execute()
               {
               for (int i = 0; i < 1; ++i)
                       {
                       IIMessage Cmd;
                       Cmd.TypeCode(ccLogin);
                       Cmd.Data(0, i);   Cmd.Data(1, 100);
```

```
        Cmd.Data(2, 101); Cmd.Data(3, 102);
        MailXfr->Send(0, Cmd);
        }
```

The first action the thread takes is to send the “Login” message to the host, to indicate the target is up and running. Note that sending a command on the target is much like sending a message on the host. A message object is 'filled in' and sent via a method call.

The values used to distinguish commands are an enumeration defined in a header that is shared between the target and the host program. This assures that both sides agree on the meaning of a message packet.

```
    while(!Terminated())
        {
        Start.Acquire();
        Open();

        Gather();
        if (Info.Plot)
            WriteBuffer(Snaps);

        Close();
        Available.Release();
        }
    }
```

Here is the thread loop. The thread remains idle until the Start semaphore is signaled. Once it is it calls the Open() method to configure the analog driver, Gather() to collect the data, and then closes the driver. The Available semaphore signals the main thread that data has been sent to the host, so it respond to commands again.

When we left the main thread, we were processing the command to take data. The DispatchPacket() function calls the CaptureThread's Acquire() method:

```
void Acquire(IIMessage & Msg)
        {
        //
        //  Fill transfer Info from message
        Info.SampleRate  = Msg.AsFloat(0);
        Info.Events      = Msg.Data(1);
        Info.Channels    = Msg.Data(2);
        Info.ClockSource = Msg.Data(3);
        Info.Plot        = Msg.Data(4);

        Info.SampleRate *= 1000;

        Start.Release();
        Available.Acquire();
        }
```

The Acquire() method copies out the information in the message into our local cache. The release of the Start semaphore wakes up the background thread. The main thread then waits for the Acquire semaphore to be signaled, suspending the main thread while data taking is going on.

In this simple example, there is no real need to respond to messages all of the time. In other cases, you might wish to continue to respond to messages and reduce the interlocking. If so, you have to take more care that a second data taking request does not corrupt the data taking information.

Configuring Data Acquisition
With the waking of the CaptureThread, Open() is called to configure the analog.

```
void Open()
        {
        ...

        AIn->Device().Channels().EnableChannels(Info.Channels);
        int IntsPerEvent = AIn->Device().Channels().BytesPerEvent() / sizeof(int);
```

Analog I/O is managed by Analog Stream objects. These use BIOS device drivers to DMA data into a set of buffer objects and return them to the application as they are filled with data. The CaptureThread uses the AdcStream object Ain which takes data from the ADCs.

Here we are configuring the number of channels to activate in the data stream, and using the information block to read back the size of the data that will come out per event.

```
// Simple, continuous data flow
BasicTmb Timebase;
Timebase.ClockSource( (Info.ClockSource==0) ?
                                ClockManager::csTimer0 : ClockManager::csDds );
Timebase.Rate(Info.SampleRate);
Ain->Device().Attach(Timebase);
```

Analog streams need to have a timebase to physically drive the I/O. These objects configure the hardware and logic to use the type of clock the application requires. The BasicTmb is a simple continuous clock. We use one of the parameters to set which physical timer is used as the timebase and attach it to the analog device.

```
EventsPerBuffer = 65536/IntsPerEvent;
int IntsPerBuffer;

// Buffers sized to signal at specified rate
Ain->Events(EventsPerBuffer);
```

The size of the buffers used to accept data is an important parameter. Too small a size creates excess overhead since you are continually switching buffers. Too large a size can either overflow memory or suspend the application waiting for the first block to finish for too long a time.

```
//  Create an image buffer for data snaps
Resize(Info.Events);
BufferCount = 1;
IntsPerBuffer = Info.Events * IntsPerEvent;
```

This application saves a fixed number of events for transmission back. The Snaps buffer is used to store this data. The Resize() method sizes it to fit the data needed.

```
volatile int status = AIn->Open();
```

The Stream's Open() method starts the acquisition of data. We then send a message to the host informing the host of the number of buffers to expect.

```
IIMessage Cmd;
Cmd.TypeCode(ccBufferCount);
Cmd.Data(0, IntsPerBuffer / MaxPayload + 1);

MailXfr->Send(0, Cmd);

FCount = 0;
```

```
        }
```

From the Open() method the Capture thread moves straight to the Gather() method. This method takes enough buffers in to fill the capture buffer. The core of the Gather() is the call to Get(), which calls Ain->Get(). This method blocks until a buffer of acquired data has arrived from the Analog input. This buffer can then be accessed as AIn->Buffer().

```
//-----------------------------------------------------
// Sample in analog buffer
//-----------------------------------------------------
void Get()
        {
        AIn->Get();
        ++FCount;
        }
```

After getting a buffer, the contents are copied into the Snaps buffer. We keep track of the location where to write the next block as the Cursor variable. When we have filled the Snaps buffer we are done.

```
//-----------------------------------------------------
// Accumulate segments into acqusition buffer
//-----------------------------------------------------
void Gather()
        {
        Cursor = 0;

        while (Cursor < Snaps.Elements())
                {
                Get();

                int Residual = Snaps.Ints()-Cursor;
                int Chunk = std::min(Residual, AIn->Buffer().Ints());

                Snaps.Copy(AIn->Buffer(), Cursor, Chunk);
                Cursor += Chunk;
                }
        }
```

After filling the Snaps buffer, we send it back to the host. Sending a data packet is similar to sending a command, except the object sent is a buffer rather than an IIMessage. There is a maximum size of packet that can be sent based on the size of the region reserved for busmastering by the device driver. The example uses a modest maximim packet size saved in the MaxPayload constant. Packets are copied out of the snap buffer in chunks of MaxPayload, until the final packet which sends the remainder.

Each data packet is preceded by a ccDataFlushMsg command. This is by no means a requirement. Data packets can be sent at any time and in any number without messages marking them, as long as both sides agree on what these data packets mean. In this example the messages allow progress feedback to be displayed as the data is acquired.

```
//-----------------------------------------------------
//  WriteBuffer()  --  Send buffer to host
//-----------------------------------------------------
void WriteBuffer(IntBuffer & Buffer)
        {
        int Elements = Buffer.Ints();
        int * Cursor = Buffer.Addr();

        while (Elements)
         {
                const int Residual = std::min(Elements, MaxPayload);

                IIMessage Cmd;
```

```
            Cmd.TypeCode(ccDataFlushMsg);
            MailXfr->Send(0, Cmd);

            // Copy buffer to payload
            IntBuffer Dst(Residual);
            IntBuffer Src(Cursor, Residual);
            Dst.Copy(Src);

            Xfr->Send(0, Dst);

            Cursor += Residual;
            Elements -= Residual;
        }
    }
```

The Close() Method
After taking the data, the Close() method is called. It sends a summary message to the host giving the results of the log session, then shuts down the data taking stream to await the next capture.

Data Logging

At this point the attention shifts to the host. The data capture operation has resulted in a sequence of messages and data packets that the host needs to process properly.

Command Packet Reception
In this example we use the notification events of the TiBusmasterStream class to alert us when a command arrives from the target. The handler was installed as part of the Open() method described earlier. On examination, the first interesting item is the argument of the event, a class named TiBusmasterStreamDataEvent.

Often an event needs to pass data to the handler. The Malibu event library restricts an event handler to have just a single argument. However, this argument can be a class that can contain any number of subelements. The class TiBusmasterStreamDataEvent contains a pointer to the stream that issued the event, which is used to extract the message from the stream.

```
//--------------------------------------------------------------------------
//  ApplicationIo::MailAvailableHandler() -- Data Packet arrival event handler
//--------------------------------------------------------------------------

void  ApplicationIo::MailAvailableHandler(TiBusmasterStreamDataEvent & Event)
{
    static int LoginTally(0);
    //
    //  Read the mail message packet
    MatadorMessage  Msg;
    Event.Sender->Recv(Msg);
```

By sending the sender pointer as data, we allow a single handler to process multiple stream object's messages. Once the message arrives, it can be analyzed and processed by a switch statement on the type code.

```
    //
    // ...Process the packet
    switch (Msg.TypeCode())
        {
        case ccInfo:
            {
```

```
            UI->AppendToLog("pmInfo");
    << additional handler code ommitted >>
```

The first message sent as part of the capture protocol was a ccBufferCount message. The purpose of this command is to inform the application of how many data buffers will be sent to complete the capture. This value is saved, and also resizes the progress bar to display the fraction of blocks remaining in the transmission.

Next comes one ccDataFlush message for each data packet, followed by the packet itself. The application actually notes these packets but displays nothing. Finally on Close() comes a final ccDataFlush command followed by a result command ccInfo. This command closes the logger, writes the data to disk, displays some of the information and creates a data description file for the BinView analysis application and saves it to disk.

Data Packet Reception

Like the message packet handler, the arrival of data packets can be processed inside of an event handler function.

```
//-----------------------------------------------------------------------------
//  ApplicationIo::PacketAvailableHandler() -- Data Packet arrival event handler
//-----------------------------------------------------------------------------

void  ApplicationIo::PacketAvailableHandler(TiBusmasterStreamDataEvent & Event)
{
    IntegerBuffer Packet;
    Event.Sender->Recv(Packet);

    UI->AppendToLog("Packet Rcvd");
    UI->UpdateAcquisitionProgress();

    std::string text = "Rx: " + IntToString(++RcvMessageTally);
    UI->MessageText(text);

    Logger.Log(Packet.IntPtr(), Packet.Size());

}
```

The sender pointer in the event argument is used to receive the data into the IntegerBuffer object Packet. After some update of the UI, the packet data is appended to data logger by the Log method. So the actual process of responding to a data packet, reading it in and logging it is accomplished in two lines of source code.

As stated above, the final ccInfo message closes the logged file, writing out the last piece of data and closing the file. This allows other programs to open and analyze the results.

Developing Host Applications

Developing an application will more than likely involve using an integrated development environment (IDE) , also known as an integrated design environment or an integrated debugging environment. This is a type of computer software that assists computer programmers in developing software.

The following sections will aid in the initial set-up of these applications in describing what needs to be set in Project Options or Project Properties.

Borland Turbo C++

BCB10 (Borland Turbo C++) Project Settings

When creating a new application with File, New, VCL Forms Application - C++ Builder

Change the Project Options for the Compiler:

Project Options
++ Compiler (bcc32)
C++ Compatibility
Check 'zero-length empty base class (-Ve)'
Check 'zero-length empty class member functions (-Vx)'

In our example Host Applications, if not checked an access violation will occur when attempting to enter any event function.

i.e.
Access Violation OnLoadMsg.Execute – Load Message Event
Because of statement
Board->OnLoadMsg.SetEvent(this, &ApplicationIo::DoLoadMsg);

Change the Project Options for the Linker:

Project Options
Linker (ilink32)
Linking – **uncheck 'Use Dynamic RTL'**

In our example Host Applications, if not unchecked, this will cause the execution to fail before the Form is constructed.

Error: First chance exception at $xxxxxxxx. Exception class EAccessViolation with message "Access Violation!"
Process ???.exe (nnnn)

Other considerations:

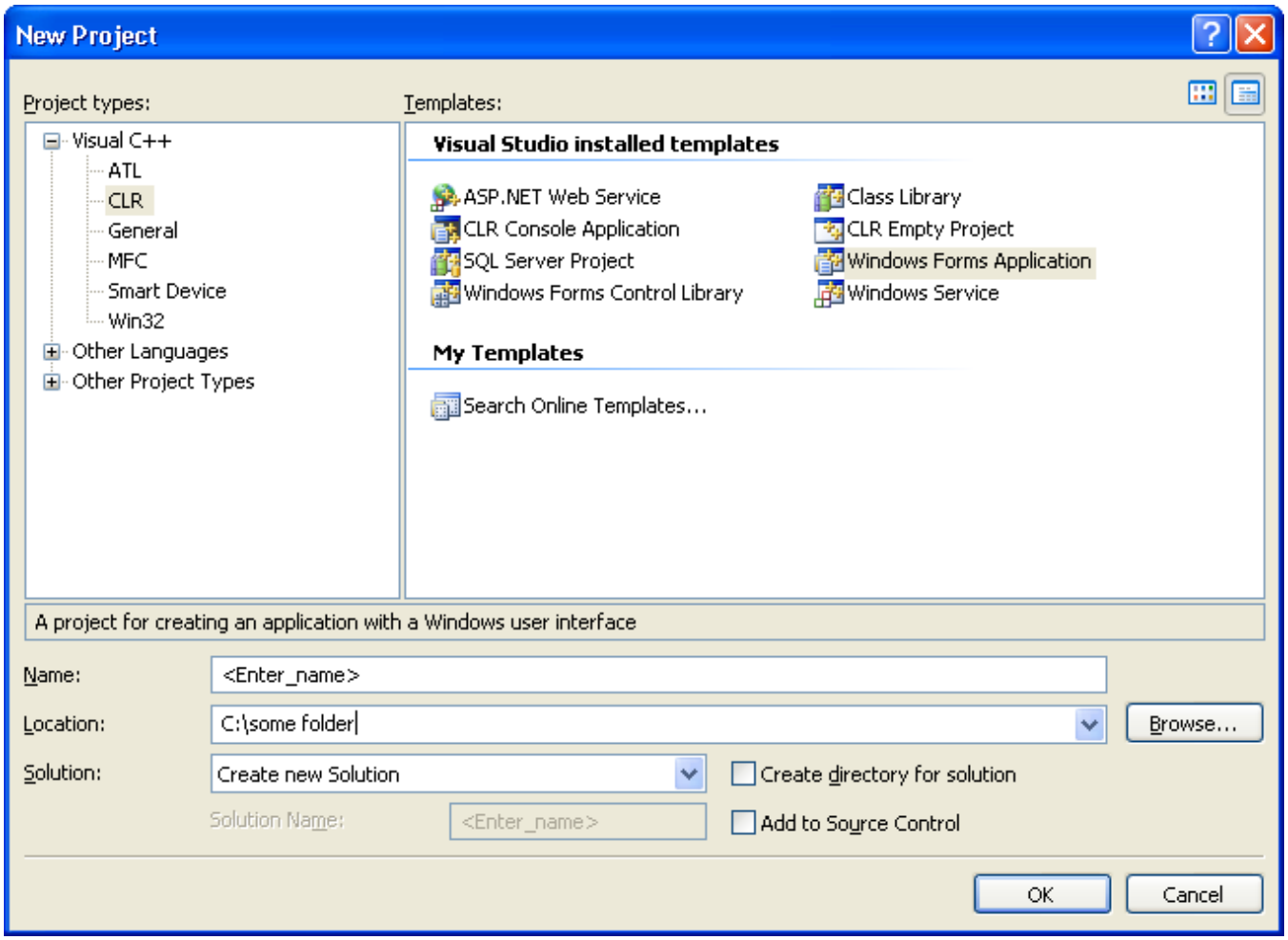
Project Options
++ Compiler (bcc32)
Output Settings
check – Specify output directory for object files(-n)
(release build) Release
(debug build) Debug
Paths and Defines
add Malibu
Pre-compiled headers
uncheck everything
Linker (ilink32)
Output Settings
check – Final output directory
(release build) Release
(debug build) Debug
Paths and Defines
(ensure that Build Configuration is set to All Configurations)
add Lib/Bcb10
(change Build Configuration to Release Build)
add lib\bcb10\release
(change Build Configuration to Debug Build)
add lib\bcb10\debug
(change Build Configuration back to All Configurations)

Packages
uncheck - Build with runtime packages

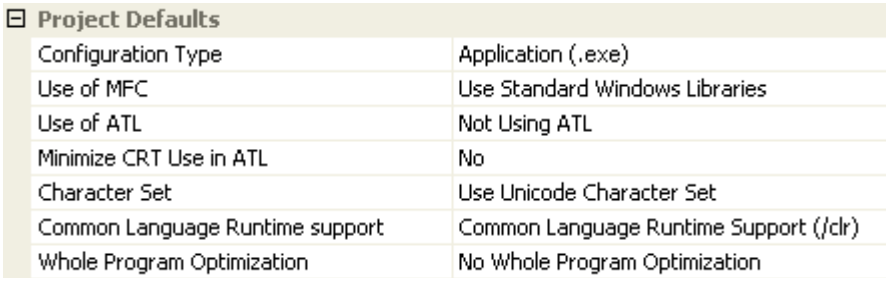
Microsoft Visual Studio 2005

Microsoft Visual C++ 2005 (version 8) Project Properties

When creating a new application with File, New, Project with Widows Forms Application:



Project Properties (Alt+F7)
Configuration Properties



Project Defaults	
Configuration Type	Application (.exe)
Use of MFC	Use Standard Windows Libraries
Use of ATL	Not Using ATL
Minimize CRT Use in ATL	No
Character Set	Use Unicode Character Set
Common Language Runtime support	Common Language Runtime Support (/clr)
Whole Program Optimization	No Whole Program Optimization

C++
General
Additional Include Directories

Malibu
PlotLab/Include – for graph/scope display

Code Generation
Run Time Library

Multi-threaded Debug DLL (/Mdd)

Precompiled Headers
Create/Use Precompile Headers

Not Using Precompiled Headers

Linker
Additional Library Directories

Innovative\Lib\Vc8

If anything appears to be missing, view any of the example sample code Vc8 projects.

Summary

Developing Host and target applications utilizing Innovative DSP products is straightforward when armed with the appropriate development tools and information.

Applets

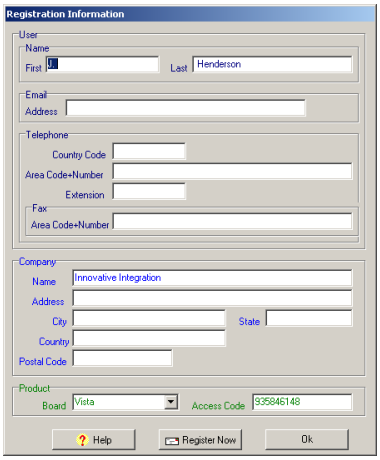
The software release for a baseboard contains programs in addition to the example projects. These are collectively called "applets". They provide a variety of services ranging from post analysis of acquired data to loading programs and logic to a full replacement host user interface. The applets provided with this release are described in this chapter.

Shortcuts to these utilities are installed in Windows by the installation. To invoke any of these utilities, go to the Start Menu | Programs | <<Baseboard Name>> and double-click the shortcut for the program you are interested in running.

Common Applets

Registration Utility (NewUser.exe)

Some of the Host applets provided in the Developers Package are keyed to allow Innovative to obtain end-user contact information. These utilities allow unrestricted use during a 20 day trial period, after which you are required to register your package with Innovative. After, the trial period operation will be disallowed until the unlock code provided as part of the registration is entered into the applet. After using the NewUser.exe applet to provide Innovative Integration with your registration information, you will receive:



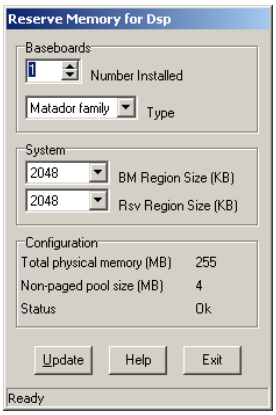
The unlock code necessary for unrestricted use of the Host applets

A WSC (tech-support service code) enabling free software maintenance downloads of development kit software and telephone technical hot line support for a one year period.

Reserve Memory Applet (ReserveMemDsp.exe)

Each Innovative PCI-based DSP baseboard requires 2 to 8 MB of memory to be reserved for its use, depending on the rates of bus-master transfer traffic which each baseboard will generate. Applications operating at transfer rates in excess of 20 MB/sec should reserve additional, contiguous busmaster memory to ensure gap-free data acquisition.

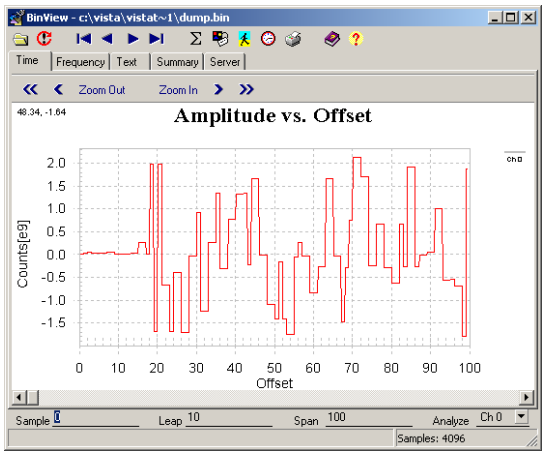
To reserve this memory, the registry must be updated using the ReserveMemDsp applet. If at any time you change the number of or rearrange the baseboards in your system, then you must invoke this applet to reserve the proper space for the busmaster region. See the Help file ReserveMemDsp.hlp, for operational details.



Data Analysis Applets

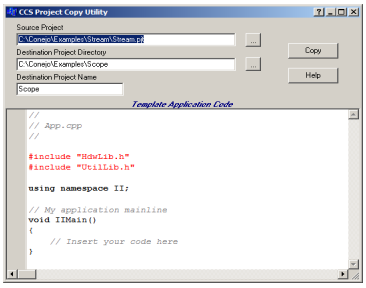
Binary File Viewer Utility (BinView.exe)

BinView is a data display tool specifically designed to allow simplified viewing of binary data stored in data files or a resident in shared DSP memory. Please see the on-line BinView help file in your Binview installation directory.



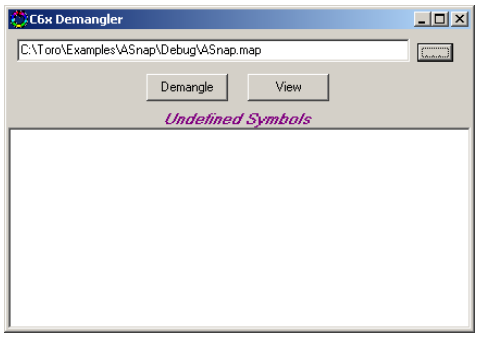
Target Project Copy Utility (CopyCcsProject.exe)

The CopyCcsProject.exe applet is used to copy all project settings from a known-good template project into a new DSP Code Composer project. This simplifies new project development, by eliminating the multi-step process of copying the myriad individual project settings from a source project in a newly-created project.



Demangle Utility (Demangle.exe)

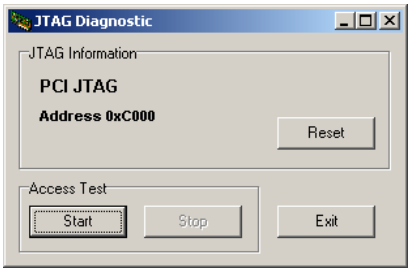
The Demangle applet is designed to simplify use of the TI dem6x.exe command-line utility. When building C++ applications, the built-in symbol mangler in the TI compiler renders symbolic names unreadable, such that missing or unresolved symbol errors displayed by the linker no longer correlate to the symbol names within your code. To work around this limitation, enable map file generation within your CCS project. Then, browse to the map file produced by the linker using the Demangle utility. The utility will display proper symbol names for all unresolved externals.



COFF Section Dump Utility (CoffDump.exe)

CoffDump.exe parses through a user-selected COFF file stored on the hard disk and ascertains the complete memory consumption by the DSP program. Memory usage for each of the sections defined in the applications command file are tabularized and the results are written to the Windows NotePad scratch buffer.

JTAG Diagnostic Utility (JtagDiag.exe)



JtagDiag.exe is used to re-initialized the JTAG scan-path interface which connects the Code Hammer debugger's PCI plug-in board with the target DSP. Use this utility prior to invoking Code Composer Studio, to insure that the communications link is viable and clear. This utility is also convenient in confirming that the Code Hammer installation is complete and correct.

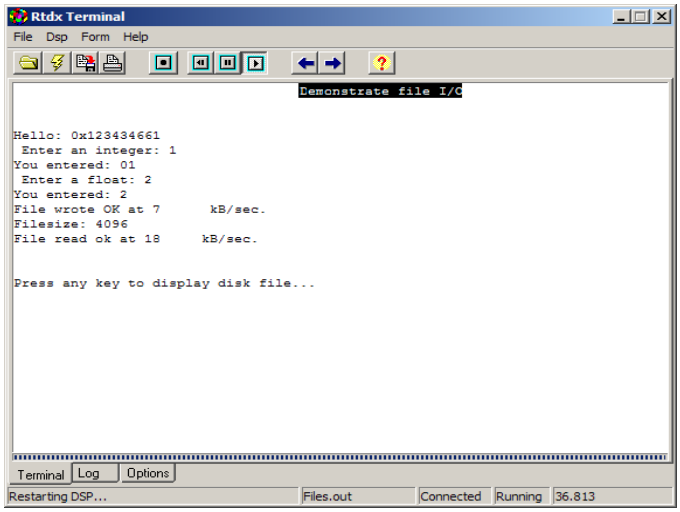
RtdxTerminal - Terminal Emulator

This applet provides a C++ language-compatible, standard I/O terminal emulation facility for interacting with the TermIo library running on an Innovative Integration target DSP processor. Display data is routed between the DSP target and this Host the terminal emulator applet in which ASCII output data is presented to the user via a terminal emulation window and host keyboard input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems, and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation during initial debugging.

RtdxTerminal is implemented as an out-of-process extension to Code Composer Studio. Consequently, it must be used in conjunction with CCS and a JTAG debugger - it cannot operate stand-alone.

The terminal emulator is straightforward to use. The terminal emulator will respond to stdio calls automatically from the target DSP card and should be running before the DSP application is executed in order for the program run to proceed normally. The DSP program execution will be halted automatically at the first stdio library call if the terminal emulator is not executing when the DSP application is run, since standard I/O uses hardware handshaking. The stdio output is automatically printed to the current cursor location (with wraparound and scrolling), and console keyboard input will also be displayed as it is echoed back from the target.

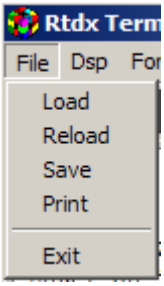
The terminal emulator also supports Windows file I/O using the TermFile library object.



Important Note:
Before using the terminal emulator, you must register your Pismo Toolset. Until you do so, usage will be restricted to a 20-day trial period for the terminal emulator and other applets contained in the Toolset. To register, fill out the contents of the Registration Form, then click on the Register Now button. This will print a Registration report which, must be faxed to Innovative Integration. Innovative Integration will E-mail you an Access Code, which must be typed into the Registration Form for all the features to be enabled.

Terminal Emulator Menu Commands
The terminal emulator provides several menus of commands for controlling and customizing its functionality. These functions are available on the menu bar, located at the top of the the terminal emulator main window. Speed button

equivalents for each of the menu options are also available on the button bar located immediately beneath the menu bar. The following is a description of each menu entry available in the terminal emulator, and its effects.



The File Menu

File | Load - provides for COFF (Common Object File Format) program downloads from within the terminal emulator. When selected, a file requester dialog box is opened and the full pathname to the COFF filename to be downloaded is selected by the user. Clicking "Open" in the file requester once a filename has been selected will cause the requester to close and the file to be downloaded to the target and executed. Clicking "Cancel" will abort the file selection and close the requester with no download taking place.

This operation can optionally be initiated via the button.

File | Reload - Reloads and executes the COFF file last downloaded to the target. It provides a fast means to re-execute the application program most recently loaded into the target board.

This operation can optionally be initiated via the button.

NOTE: File | Load and File | Reload functions use the JTAG debugger and Code Composer Studio in order to effect the program download.

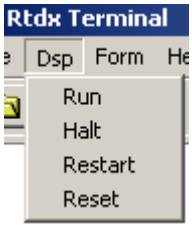
File | Save – saves the textual contents of the Terminal and Log tabs to a user specified file.

File | Print - prints the textual contents of the Terminal and Log tabs to a user specified printer.

File | Exit – closes the emulator application, terminating console emulation.

The DSP Menu

Dsp | Run - causes the terminal emulator to bring the target board into a cold-start, uninitialized condition. This is functionally identical to performing Debug | Run within Code Composer Studio.



This operation can optionally be initiated via the button.

Dsp | Halt - causes the terminal emulator to suspend DSP program execution. This is functionally identical to performing Debug | Halt within Code Composer Studio.

This operation can optionally be initiated via the button.

Dsp | Restart - rewinds the DSP program counter to the application entry point, usually *c_int00()*. This is functionally identical to performing Debug | Restart within Code Composer Studio.

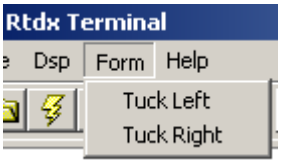
This operation can optionally be initiated via the button.

Dsp | Reset - causes the terminal emulator to bring the target board into a cold-start, uninitialized condition. This is functionally identical to performing Debug | Reset Dsp within Code Composer Studio.

This operation can optionally be initiated via the button.

The Form Menu

Form | Tuck Left - repositions the main application window to the bottom left of the Windows desktop.



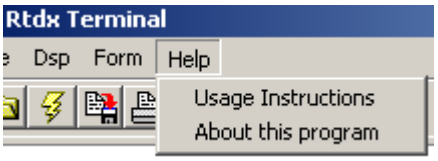
This operation can optionally be initiated via the button.

Form | Tuck Right - repositions the main application window to the bottom right of the Windows desktop.

This operation can optionally be initiated via the button.

The Help Menu

Help | Usage Instructions - displays online help detailing use of the application, including command-line arguments.



This operation can optionally be initiated via the button.

Help | About this Program - displays a dialog containing program revision and tech support contact information.

Options Tab:

The Options tab (seen below) contains controls to allow user-customization of the appearance and operation of the terminal emulator.

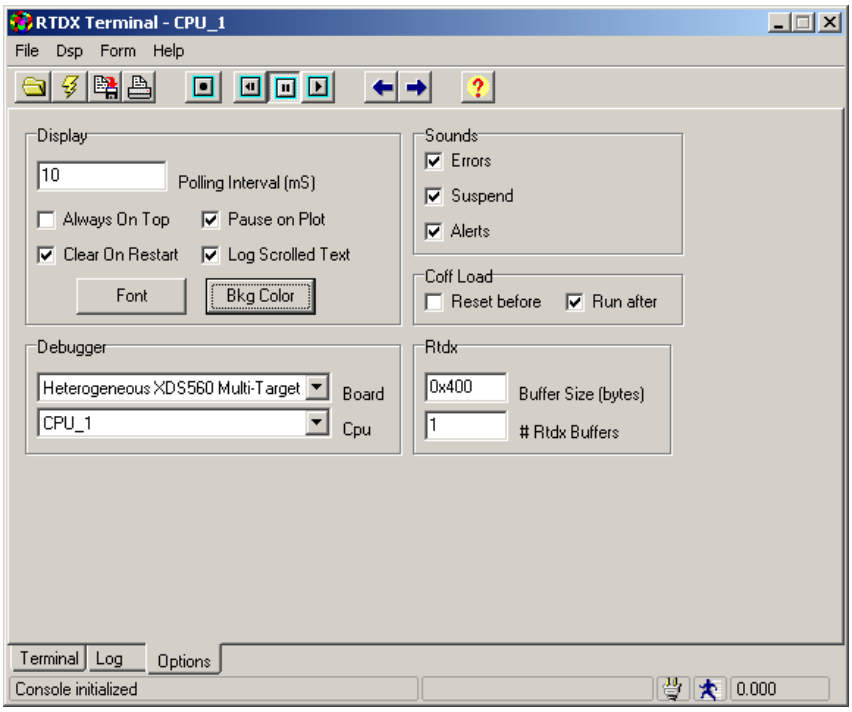


Figure 10. RtdxTerminal Options

Display *Group*
Controls within the Display group box govern the visual appearance of the terminal emulator, as detailed below.

Polling Interval - specifies the period, in milliseconds, between queries for data received from the DSP via the JTAG RTDX interface. Lower numbers increase performance but increase Host CPU load.

Always on Top - specifies that the terminal application should always remain visible, atop other applications on the Windows desktop. This check box controls whether the terminal emulator is forced to remain a foreground application, even when it loses keyboard focus. This is useful when running stdio-based code from within the Code Composer environment, when it's preferable to make terminal visible at all times. The terminal will remain atop other windows when this entry is checked. Select the entry again to uncheck and allow the terminal emulator window to be obscured by other windows.

Clear on Restart - specifies whether the terminal display and log will be automatically cleared whenever the DSP is restarted.

Pause on Plot - specifies whether standard I/O will be suspended following display of graphical information in the Binview applet which is automatically invoked via use of the Pismo library

Plot() command. If enabled, standard I/O may be resumed by clicking the button.

Log Scrolled Text - specifies whether text information which scrolls off screen on the Terminal tab is appended to the Log display. If enabled, standard I/O performance will degrade slightly during lengthy text outputs.

Font - button invokes a font-selection dialog which allows selection of user-specified font within the Terminal and Log text controls.

Bkg Color - button invokes a color-selection dialog which allows selection of user-specified background color within the Terminal and Log text controls.

Sounds Group
Controls within the Sounds group box govern the audible prompts generated by the terminal emulator, as detailed below.

Errors - if enabled, file I/O and other errors encountered during operation generate an audible tone.

Suspend - if enabled, suspension of standard I/O, such as following plotting via Binview, generate an audible tone.

Alerts - if enabled, alert conditions encountered during standard I/O, such as upon display of the ASCII bell character, generate an audible tone.

Coff Load Group
Controls within the Coff Load group box govern behavior surrounding a COFF executable download.

Reset Before - if enabled, the Code Composer Debug | Reset DSP behavior is executed before attempting to download the user-specified COFF file.

Run After - if enabled, the Code Composer Debug | Run behavior is executed immediately following the download of a user-specified COFF file.

Debugger Group
Controls within the Debugger group box specify the target DSP with which RTDX communications is established.

Board - specifies the board hosting the target DSP to be used in RtdxTerminal stdio communications. This combo box is populated with all available board types configured using the Code Composer Setup utility.

Cpu - specifies the identifier of the specific DSP to be used in RtdxTerminal stdio communications. This combo box is populated with all available CPUs present on the baseboard as configured using the Code Composer Setup utility.

Terminal Emulator Command Line Switches
The terminal emulator also provides the following command line switches to further modify program behavior. The switches must be supplied via the command line or within Windows shortcut properties (see the Installation section for more information), and will override the default behavior of the applet.

Multiple instances of the terminal emulator may be invoked simultaneously in order to support installations utilizing multiple target boards. Instances of the terminal emulator, after the first loaded instance must be configured via command line switches in order to properly communicate with their associated target.

-board *boardtype* - Use the -board switch to force an instance of the terminal emulator to communicate with a specific type of target board, boardtype. Supported board types are those configured using the Code Composer Setup utility, such as "C64xx Rev 1.1 XDS560 Emulator".

-cpu *cputype* - Use the -cpu switch to force an instance of the terminal emulator to communicate with a specific CPU on a target board. Supported CPU types are those configured using the Code Composer Setup utility, such as "CPU_1" or "CPU_A".

-f *filespec* - Use the -f switch to force the terminal emulator to load and run the specified COFF file. The "*filespec*" field should be a standard Windows file specification, including both the path and file name as a unit, to allow the user to force the terminal emulator to download the specified file to the target DSP board, as soon as the terminal emulator is loaded. This field is particularly useful in situations where the the terminal emulator is "shelled to" from within an other Host applications to facilitate the automatic execution of target applications employing standard I/O.

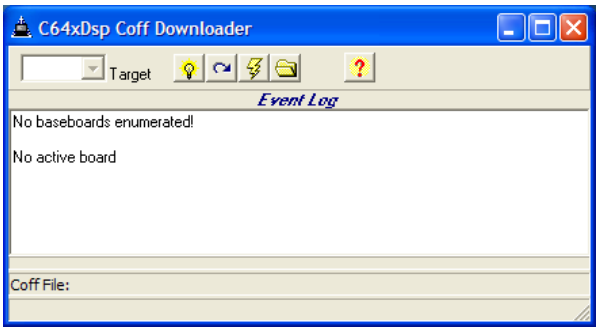
Applets for the C64x DSP Processor

The C64x Processor appears on the PCI Bus as an independent object from the baseboard it resides on. These Applets use this C64x baseboard alone and can be used on any baseboard that possesses a C64x DSP on it.

COFF Downloader (Download.exe)

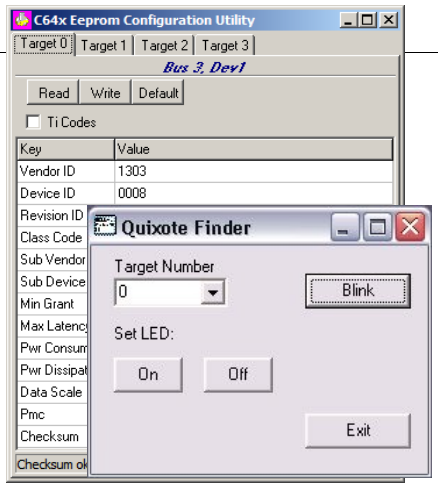
The download applet is used to load known-operational DSP executables to DSP baseboards.

The utility may be used to start DSP applications on PC power-up, through its command-line interface, or to start a DSP application from its GUI Windows user interface. It is also capable of downloading a minimal "boot" application, which is convenient when attempting to start a new Code Composer debug session after having initialized the JTAG scan path with JtagDiag.exe.



ConfigRom: C64x DSP EEProm Configuration Utility (C64xEeprom.exe)

The C64xEeprom applet is used to program the configuration values which are queried by the operating system during enumeration. These values are very rarely changed, and should only be altered under the direction of an Innovative Integration technician.



Board Finder (Finder.exe)

The board Finder is designed to help correlate baseboard target numbers against PCI slot numbers in systems employing multiple baseboards.

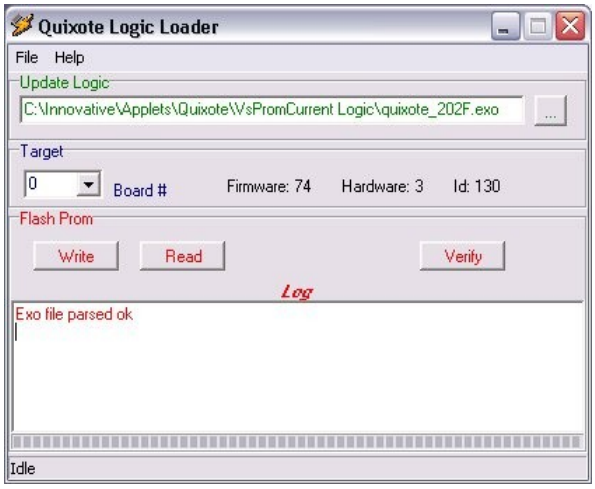
Target Number
Select the Target number of the baseboard you wish to identify using the Target Number combo box.

Blink
Click the Blink button to blink the LED on the baseboard for the specified target. It will continue blinking until you click Stop.

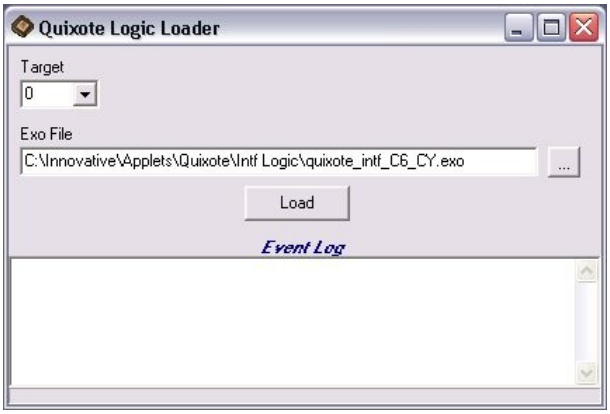
On/OFF
Use the On and Off buttons to activate or deactivate (respectively) the LED on the baseboard for the specified target. When you exit the application, the board's LED will remain in the state programmed by this applet.

PCI Logic Update Utility (VsProm.exe)

The Quixote baseboard has two logic devices on it. One controls the analog hardware and the interface to the DSP. This logic can be modified by the user, and must be loaded by the user with an image on each session. The second device performs the baseboard enumeration and PCI interface and has a ROM so that it can function at power up. The PCI Logic Update Utility applet is designed to allow field-upgrades of this PCI logic firmware on the Quixote baseboards. The utility permits an embedded firmware logic update file to reprogrammed into the baseboard Flash ROM, which stores the "personality" of the board. Complete functionality is supplied in the application's help file.



User Logic Download Utility (QuixoteLogicLoader.exe)



The logic download applet is used to deliver known-operational logic images to the user logic device installed on a Quixote baseboard. The user logic must be loaded once per session, as the logic part is cleared on bus reset or power up.

The utility may be used to configure the firmware either through its command line interface or from its GUI Windows user interface. The former is often convenient during PC boot-up to install a standard logic file. Place a short cut with the command line option set into the Windows Startup folder to execute the program when the system is started.

This application supports configuration of the onboard Virtex logic device from an EXO file produced by popular logic design tools (including Xilinx's). It is essential that the Virtex be programmed before attempting to download COFF images to the DSP, since some of the baseboard peripherals are dependent on the personality of the configured logic.

Custom Logic Development

Signal processing applications such as software-defined digital radio require very high speed operations for filtering, channel separation and down-conversion prior to symbol decoding. While the ‘6416 is a very powerful signal processor, when the processing rates exceed a few MHz, it is necessary to perform the signal processing in the FPGA. In the FPGA it is possible to process data at very high rates, up to several hundred MHz, because of the highly parallel and focused architectures that can be implemented. In the case of a wireless application, the signal processing is divided into high speed functions, such channel separation and down-conversion performed on the FPGA, and low speed operations, such as symbol decoding and data formatting performed on the DSP.

Begin by analyzing your application to identify the operations that are high speed (above 1 MHz) and lower speed. Higher speed signal processing operations should be targeted at the FPGA provided that they are of manageable complexity. Typical FPGA operations include FIR filters, down-conversion, specialized high speed triggering and data sampling, and FFTs. All of these functions are deterministic mathematical functions that are suitable for the FPGA. Data formatting, protocols and control functions are typically more easily implemented on the DSP.

The Quixote has two FPGAs: a Xilinx Spartan2 (200K gates) and a Xilinx Virtex2 (2M or 6M gates). The Virtex2 is used for the analog interfacing, and as the computational logic on the board. Major elements inside the Spartan2 logic are the PCI interface, interrupt control, timebase selection and message mailboxes. The Spartan2 source code is not normally provided.

Quixote II and III are open logic products that include additional memory for FPGA computational space, and either a 2M (II) or 6M (III) gate Xilinx Virtex2 FPGA. The logic in Quixote I is provided in source form to assist in custom application developments.

Quixote supports in-circuit reconfiguration of the Xilinx Virtex2 over its Select Map interface. FPGA logic images may be downloaded to the Virtex2 logic as part of the software application at any time of its SelectMap interface. The Mailbu toolset provides software support for logic configuration, and logic downloading applets are also provided in the software release. See the Applets chapter for more details.

Quixote also a Xilinx Spartan2 for the streaming PCI interface and infrastructure that has a non-volatile logic image, loaded by hardware when the card powers up. The Spartan logic is also field upgradeable using the supplied applet in the software release.

Overview

The Quixote Logic Framework is the basis for custom logic application development. This Framework allows the FPGA designer to begin with a known working design that illustrates Analog I/O and DSP interfacing techniques used in Quixote I. The custom application is then fit within the framework and includes the application-specific code. In many cases, the standard methods for moving data to the DSP are retained so that custom software development is minimized.

Target Devices

Quixote has as its user programmable logic a Xilinx Virtex2 FPGA. The Virtex2 FPGAs have many resources in addition to the logic gates such as embedded RAM blocks, digital clock managers (DCMs), and flexible IO standards that are used in the sample logic. The Quixote is offered with either a 2M gate device or 6M gate device.

Table 19. Logic Devices on Quixote Variants

Quixote Variant	Logic Density	Device Used
Quixote II (80069-2)	2M	Xilinx XC2V2000-4FF896C
Quixote III (80069-3)	6M	Xilinx XC2V6000-4FF1152C

Note: Quixote I (80069-1) does not support user programming of the logic.

Code Development Tools

Developing logic code for this FPGA is done using the Xilinx ISE Toolset and the Framework logic provided. The Framework logic is delivered in VHDL language, along with the control files for the Xilinx ISE software to allow the user to modify and recompile the logic. This Framework logic gives minimal interface and control for the various peripherals and is provided only as a means to get started coding with FPGA.

Simulation files are provided for use with Mentor Graphics ModelSim. A macro for the testbench (tb_quixote.do) is provided to compile and load the simulation. When using ModelSim, it is required to compile and reference the library files for unisim, simprim, and xilinxlogiccore for the simulation to load properly.

Steps for Successful Use

- You must have a simulation tool. Innovative provides files for ModelSim. If you don't simulate, it is unlikely you will successfully complete the logic design within your lifetime.
- Estimate the logic usage for your application. For reference, the Framework is about 11% of the 6M device. Of particular importance are the limited resources such as DCMs and blockRAMs. A little planning saves time later.
- Define the clock domains in the design and methods to transition them. FIFOs work great.
- Identify crucial timing constraints. Plan the logic method to meet them. Pipelining is usually a good approach.
- Until your simulations work, the hardware will not work. Debugging on real hardware is very time consuming. Perform real timing simulations wherever possible.

Using the FPGA Framework

As a starting point, here is a big overview of what is attached to the Virtex 2 FPGA. All of the high speed data interfaces have 32K sample FIFOs for buffering the data to the Virtex2 logic. The DSP has both buses attached to the logic, and also two of the McBSP ports so that ample data movement is supported.

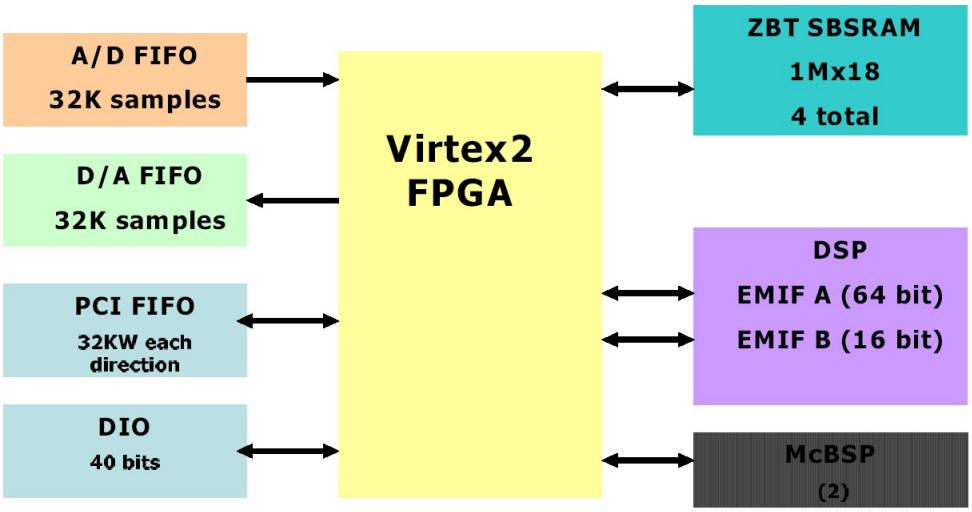
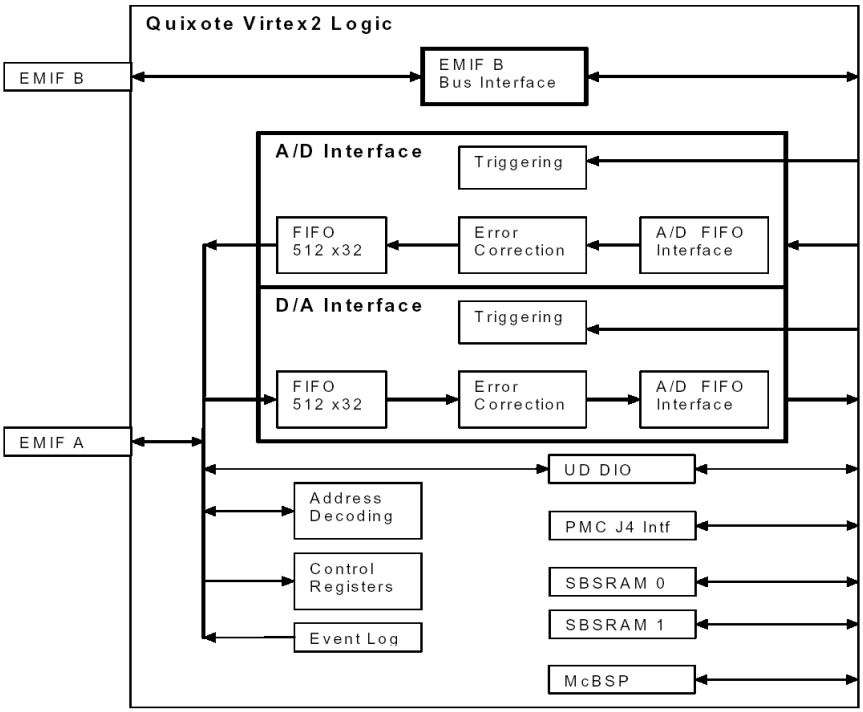


Figure 11. Overview of FPGA External Connections

The Framework Logic block diagram is shown below.

As can be seen in the block diagram, the EMIF A is used as the primary interface to the analog IO. The EMIF A memory space is decoded so that control and configuration registers are mapped as well as burst FIFOs for the high speed data paths. EMIF B is used primarily as the PCI interface bus, and travels through the Virtex2 en route to the Spartan2 logic. Two McBSP interfaces are also provided.



Memory mapped peripherals reside on both EMIFs in the Logic Framework that define the bus interface type and timing parameters. A memory map for the Framework Logic is included below. Most designers integrate application-specific features into the standard memory mapping to preserve as much software as possible. Also, the memory decoding and data interface to the DSP are architected in the logic to support the burst or asynchronous memory interfaces.

Table 20. Quixote Memory Map

CE Space	Address (Hex)	Description	Logic Device	R/W	CE Type
BCE0	0x60000000	PCI FIFO	Spartan2	R/W	Burst
BCE1	0x64010000	Readback Register (for test)	Spartan2	R/W	Async
	0x64030000	Timebase PLL control	Spartan2	W	Async
	0x64040000	DSP status	Spartan2	R	Async
	0x64050000	Mailbox Command Register	Spartan2	R/W	Async
	0x64060000	Mailbox Command Clear	Spartan2	W	Async
	0x64070000...3C	Mailbox RAM	Spartan2	R/W	Async
	0x64080000	NMI Status/Ack	Spartan2	R/W	Async
	0x64090000	Int Source Status	Spartan2	R	Async
	0x64090000	Int4 Source Enable	Spartan2	W	Async
	0x640A0000	Int5 Source Enable	Spartan2	W	Async
	0x640B0000	Int6 Source Enable	Spartan2	W	Async
	0x640C0000	Int7 Source Enable	Spartan2	W	Async
	0x640D0000	NMI Enable	Spartan2	W	Async
	0x640E0000	Cal EEPROM Clock	Spartan2	W	Async

CE Space	Address (Hex)	Description	Logic Device	R/W	CE Type
	0x640F0000	Cal EEPROM data	Spartan2	R/W	Async
	0x64100000	Int Type	Spartan2	W	Async
	0x64110000	Int Polarity	Spartan2	W	Async
	0x64120000	PXI configuration	Spartan2	W	Async
	0x64130000	Convert Configuration	Spartan2	W	Async
	0x64140000	Timer Configuration	Spartan2	W	Async
	0x64150000	External PCI FIFOs control	Spartan2	W	Async
BCE2	0x68010000	Not used	-	-	-
BCE3	0x6C000000	Virtex2 Test Register	Virtex2	R/W	Async
ACE0	0x80000000	A/D control	Virtex2	W	Async
	0x80020000	A/D burst Length	Virtex2	W	Async
	0x80040000	Not used	-	-	
	0x80060000	Event Log Enables	Virtex2	W	Async
	0x80080000	D/A control	Virtex2	W	Async
	0x800A0000	Not used	-	-	-
	0x800C0000	Not used	-	-	-
	0x800E0000	DAC Burst Length	Virtex2	W	Async
	0x80100000	A/D Start Trigger Selection	Virtex2	W	Async
	0x80120000	A/D Start Trigger Configuration	Virtex2	W	Async
	0x80140000	A/D stop Trigger Selection	Virtex2	W	Async
	0x80160000	A/D Stop Trigger Configuration	Virtex2	W	Async
	0x80180000	A/D Trigger Threshold	Virtex2	W	Async
	0x801A0000	A/D Threshold Channel Enable	Virtex2	W	Async
	0x801C0000	A/D Frame Timer Configuration	Virtex2	W	Async
	0x801E0000	A/D Frame Count Configuration	Virtex2	W	Async
	0x80200000	D/A Start Trigger Selection	Virtex2	W	Async
	0x80220000	D/A Start Trigger Configuration	Virtex2	W	Async
	0x80240000	D/A Stop Trigger Selection	Virtex2	W	Async
	0x80260000	D/A Stop Trigger Configuration	Virtex2	W	Async
	0x80280000	Not used	-	-	-
	0x802A0000	Not used	-	-	-
	0x802C0000	D/A frame timer Configuration	Virtex2	W	Async
	0x802E0000	D/A Frame Counter Configuration	Virtex2	W	Async
	0x80300000	UD configuration	Virtex2	W	Async

CE Space	Address (Hex)	Description	Logic Device	R/W	CE Type
	0x80320000	Analog Configuration Register	Virtex2	W	Async
	0x80340000	Analog Compare DAC control	Virtex2	W	Async
	0x80360000	Not used	-	-	-
	0x80380000	SBSRAM 0 Address	Virtex2	W	Async
	0x803A0000	SBSRAM 1 Address	Virtex2	W	Async
	0x803C0000	Not used	-	-	-
	0x803E0000	A/D Gain Memory	Virtex2	W	Async
	0x80400000	A/D Offset Memory	Virtex2	W	Async
	0x80420000	D/A Gain Memory	Virtex2	W	Async
	0x80440000	D/A Offset Memory	Virtex2	W	Async
	0x80460000	Not used	-	-	-
	0x80480000	Not used	-	-	-
	0x804A0000	Not used	-	-	-
	0x804C0000	PCI FIFO burst read length	Virtex2	W	Async
	0x804E0000	PCI FIFO burst write length	Virtex2	W	Async
	0x80500000	UD Data	Virtex2	R/W	Async
	0x80520000	SBSRAM 0 Data	Virtex2	R/W	Async
	0x80540000	SBSRAM 1 Data	Virtex2	R/W	Async
	0x80560000	PMC J4 test port	Virtex2	R	Async
	0x80580000	Misc Status	Virtex2	R	Async
	0x805A0000	A/D FIFO status	Virtex2	R	Async
	0x805C0000	Event Log	Virtex2	R/W	Async
ACE1	0x90000000	A/D FIFO data	Virtex2	R	Burst
	0x90000000	D/A FIFO data	Virtex2	W	Burst
ACE2	0xA000000	Not used	-	-	-
ACE3	0xB000000	Not used	-	-	-

Developing FPGA Firmware

Virtex2 designs for the Quixote are VHDL files and were compiled under Xilinx ISE 6, XST synthesis. The example/debug logic design is a project loadable by opening ISE and performing a Project | Open on the quixote_intf.npl file in the logic root directory. The design directory path originally used to house this project hierarchy was C:\PROJECTS\CQUIXOTE\REVB\LOGIC. It is recommended that the customer's installation duplicate this directory configuration in order to avoid problems with the Xilinx tools.

Setup for the environment is included in the npl file. Constraints for the example logic are quix_intf_ff896.ucf for the 2M device or quix_int_ff1152.ucf for the 6M device and should be applied to the quixote_intf.vhd file, which is the top of the design. The constraints have the pin assignments, IO standards, and some basic timing constraints for the framework. Compilation is usually easy, and users familiar with the Xilinx ISE can start by verifying that the design recompiles. Just click on the synthesis, then Implement, then generate programming file buttons to get through the design process. Watch out for the warnings, such as timing constraint violations, as these may really mean your design won't work at speed. These are viewed in the report files. The tools issue many warnings along the way, these can be ignored for the sample logic.

Note: If you are unfamiliar with Xilinx ISE, see the Xilinx tutorial on using it, accessed from the help button on the top of the application.

Once you have successfully compiled and installed the software, your development can begin by editing out code you don't need and deleting the associated constraints. You will have to keep the constraints associated with all pin assignments whether or not you use that function as pins are fixed by the board design.

Steps to Follow for Logic Builds

- Install Xilinx ISE 6.0 or above.
- Unzip the Framework Logic into a suitable directory.
- Open the ISE project quixote_intf_c.npl.
- Version ISE 6.2 Notes : ISE will report that the project file needs updating which should be OK. It may also be necessary to regenerate the DDC core under the current CoreGen tool, changing the filter coefficient files (.coe files) to point at your local directory structure.
- Recompile and refit the logic. The resultant output file will be quixote_intf.bit. Intermediate report files are quixote_intf.bld (NGD Build report), quixote_intf.mrp (MAP report), and quixote_intf.par (Place and Route Report). The .PAR file shows a timing summary. None of these files should include errors, only warnings. The warnings can be safely ignored.
- Generate a PROM image as an .exo file using Xilinx Impact. Use one bitstream, starting at 0x0, with Motorola S-Record (EXO) format for the bit file quixote_intf.bit. The PROM size my be set to auto.
- For the first time, compare the file generated to the provided .exo for the for your target (2M or 6M device) using a file compare program. If the compare is identical, you have successfully regenerated the logic image and are ready to proceed.

Logic Files and Hierarchy

Within the ISE environment, the logic for the Quixote has a structure as shown in the figure below.

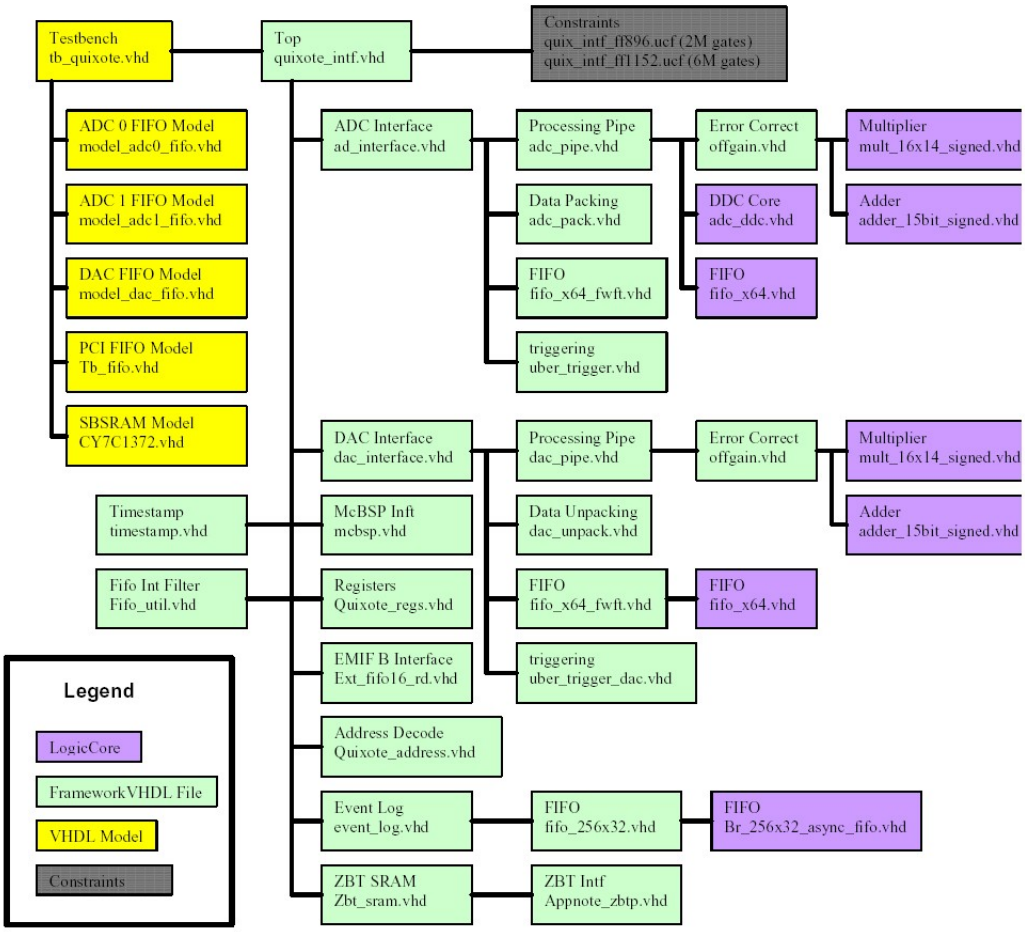


Figure 12. Logic Structure and Files

The constraints file, quix_intf_ff896.ucf, controls the pin mappings for the 2M gate FPGA (XC2V2000- 4FF896C) and timing constraints. For the 6M gate part (XC2V6000-4FF1152C), use quix_intf_ff1152.ucf.

Fitting Results for the Framework Logic

The Framework logic consumes about 11% of the logic for the 6M device. The results of the mapping process, as taken from the quixote_intf.mrp report, are as shown. Notice the memory consumed during the compile is 291 MB for this design!

```
Design Summary
--------------
Number of errors:      0
Number of warnings:   66
Logic Utilization:
  Number of Slice Flip Flops:      7,892 out of 67,584     11%
  Number of 4 input LUTs:          5,253 out of 67,584      7%
Logic Distribution:
    Number of occupied Slices:                         5,138 out of 33,792 15%
    Number of Slices containing only related logic:    5,138 out of 5,138 100%
    Number of Slices containing unrelated logic:           0 out of 5,138 0%
        *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:          6,410 out of  67,584     9%
      Number used as logic:                      5,253
      Number used as a route-thru:                 435
      Number used as Shift registers:              722

    Number of bonded IOBs:            541 out of     824    65%
       IOB Flip Flops:                              554
       IOB Master Pads:                               3
       IOB Slave Pads:                                3
    Number of Tbufs:                   40 out of  16,896     1%
    Number of Block RAMs:               9 out of     144     6%
    Number of MULT18X18s:              12 out of     144     8%
    Number of GCLKs:                    3 out of      16    18%
    Number of DCMs:                     2 out of      12    16%

    Number of RPM macros:         8
Total equivalent gate count for design:  817,570
Additional JTAG gate count for IOBs:  25,968
Peak Memory Usage:   291 MB
```

Figure 13. Map Results

The Place and Route Step results are taken from the quxiote_intf.par report. Timing analysis is shown for the design. It is important to review this report to find timing errors. The PAR report shows how the design performed against each defined timing constraint. For further analysis, the timing analyzer tool can be used to pinpoint the source of each problem.

```
•   The Number of signals not completely routed for this design is: 0

•      The Average Connection Delay for this design is:        1.773 ns
•      The Maximum Pin Delay is:                             14.941 ns
•      The Average Connection Delay on the 10 Worst Nets is:   9.396 ns

•      Listing Pin Delays by value: (ns)

•       d < 3.00   < d < 6.00   < d < 9.00   < d < 12.00  < d < 15.00  d >= 15.00
•       ---------  ---------    ---------    ---------    ---------    ---------
•           32225       5805         1719           19            1            0

•   Timing Score: 102

•   WARNING:Par:62 - Timing constraints have not been met.

•   Asterisk (*) preceding a constraint indicates it was not met.

•   ------------------------------------------------------------------------------
•     Constraint                              | Requested  | Actual     | Logic
•                                             |            |            | Levels
•   ------------------------------------------------------------------------------
```

```
•     NET "dac_clk_buf" PERIOD =  10 nS  HIGH  | 10.000ns   | 7.334ns     | 1
•     50.000000 %                             |            |             |
•   ------------------------------------------------------------------------------
•   * NET "adc_clk_buf" PERIOD =  10 nS  HIGH  | 10.000ns   | 10.102ns    | 13
•     50.000000 %                             |            |             |
```

Figure 14. PAR Results

Adding functionality to the Framework Logic

The framework logic is a starting point for the more advanced logic that will be your Quixote applicAtion logic. It is suggested that you begin by simply recompiling this logic and verifying that you can recreate the framework logic as delivered (see steps above). This will verify that you have all the libraries and FPGA compilation tools required to move ahead on your design.

Once you have successfully recompiled the logic, it is now possible to begin adding and replacing the simple logic with your application code. This is done by modifying the top VHDL to include your sub-functions, then modifying the test bench code to adequately stimulate your design.

Innovative Integration strongly recommends that you fully simulate your design before putting this logic into the Quixote FPGA. This will not only save time in debugging, but could also prevent simple errors from causing serious damage to the module. A tool like ModelSim is generally required for this high density, complex logic design that give full visibility into the logic behavior prior to actual synthesis.

Many pre-written logic functions are available to assist in logic development from Xilinx and other vendors. These logic functions may be viewed at the Xilinx web site (http://www.xilinx.com/ ipcenter/index.htm). These logic functions include basic math, filters, FFTs, SBSRAM controllers, NCOs and a multitude of other functions that are useful in logic designing with the Xilinx Virtex2 FPGAs.

Logic Design Methods and Code Development

DAC Data Path

The Framework Logic implements a data path for the DACs as shown in the following block diagram. The functions implemented in the logic are shown in the violet logic box, while the other devices are external hardware. The DAC consumes data from the external FIFO whenever the trigger is true and a timebase is enabled. The external FIFO REN (read enable) signal is true whenever the trigger is active.

Data flow is metered into the external FIFO using the level flag ALMOST FULL (AF). Whenever AF is false, data is demanded by the process from the data bus FIFO and is flowed through the data pipe and error correction to the external FIFO. As can be seen, the data path originates from DSP EMIF A bus as a 64-bit wide interface. Data is simply stacked on the bus as ch1/ch0/ch1/ch0 for efficiency. This data is then unstacked as required by the data process and run through the error correction to the external FIFO. A DMA channel is attached to the FIFO that writes data to the FIFO from your buffer memory whenever data is available, as defined by the programmable threshold.

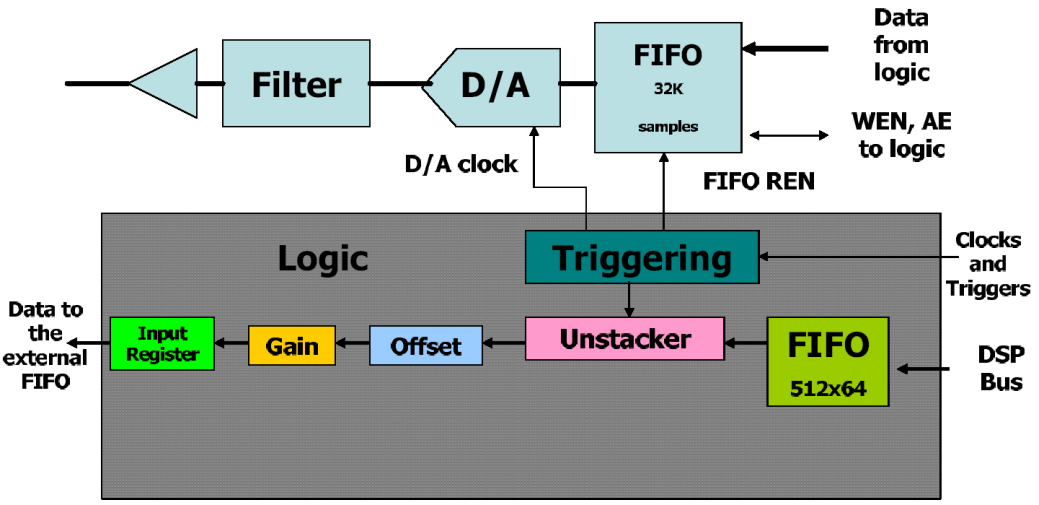


Figure 15. DAC Interface Block Diagram

A/D Data Path

The Framework Logic implements a data path for the A/Ds as shown in the following block diagram. The functions implemented in the logic are shown in the violet logic box, while the other devices are external hardware. The A/D delivers data to the external FIFO whenever the trigger is true and a timebase is enabled. The external FIFO WEN (write enable) signal is true whenever the trigger is active.

Data flow is metered from the external FIFO based upon the level of the internal FIFO. Whenever the FIFO in the logic has space, data is flowed through the data pipe and error correction to the logic FIFO. As can be seen, the data path to DSP EMIF A bus as a 64-bit wide interface. Data is simply stacked on the bus as ch1/ch0/ch1/ch0 for efficiency for the EMIF A reads. A DMA channel is attached to the FIFO that reads the FIFO whenever data is available, as defined by the programmable threshold.

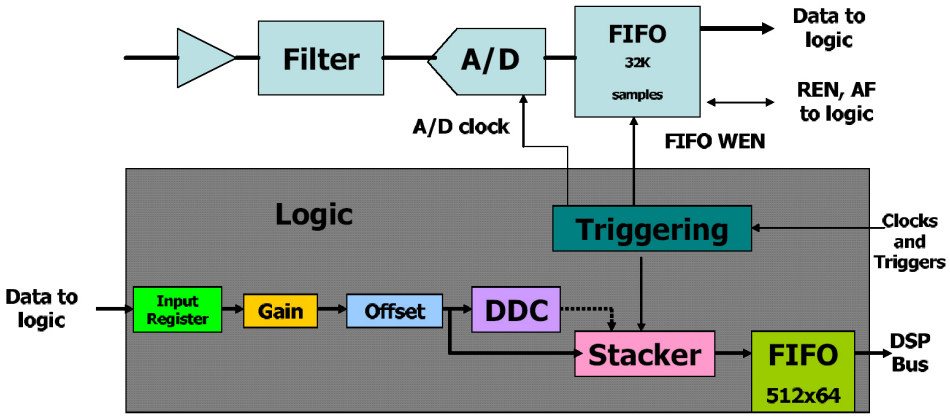


Figure 16. A/D Interface Block Diagram

DSP Interface Logic

The Quixote EMIF A memory decodings are arranged so that ACE0 is asynchronous and ACE1 is synchronous burst memory. Generally speaking, the async peripherals are not used as high speed devices since this is inherently a slow access protocol - the sync burst memory interface is at least 4x faster. The following diagrams show the DSP access timing for burst reads and writes.

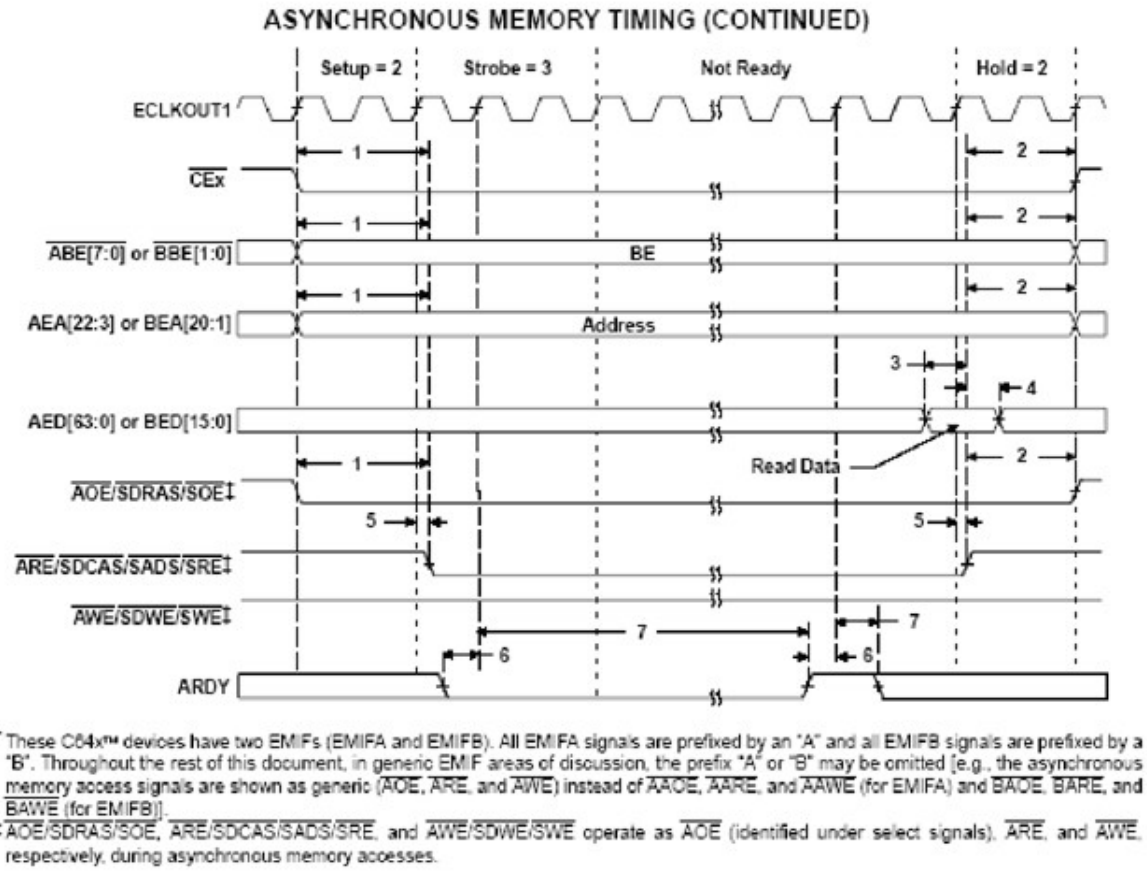

ASYNCHRONOUS MEMORY TIMING (CONTINUED)
Setup = 2
Strobe = 3
Not Ready
Hold = 2
ECLKOUT1
CEx
ABE[7:0] or BBE[1:0]
BE
AEA[22:3] or BEA[20:1]
Address
AED[63:0] or BED[15:0]
Read Data
AOE/SDRAS/SOE‡
ARE/SDCAS/SADS/SRE‡
AWE/SDWE/SWE‡
ARDY
1
2
3
4
5
6
7
† These C64x™ devices have two EMIFs (EMIFA and EMIFB). All EMIFA signals are prefixed by an "A" and all EMIFB signals are prefixed by a "B". Throughout the rest of this document, in generic EMIF areas of discussion, the prefix "A" or "B" may be omitted [e.g., the asynchronous memory access signals are shown as generic (AOE, ARE, and AWE) instead of AAOE, AARE, and AAWE (for EMIFA) and BAOE, BARE, and BAWE (for EMIFB)].
‡ AOE/SDRAS/SOE, ARE/SDCAS/SADS/SRE, and AWE/SDWE/SWE operate as AOE (identified under select signals), ARE, and AWE, respectively, during asynchronous memory accesses.

Figure 17. DSP Asynchronous Timing. (Courtesy of Texas Instruments)

For slow speed devices such as configuration and control registers, asynchronous access are used in the Framework Logic. Asynchronous accesses provide the most flexibility in timing control and are the easiest to use in most designs, albeit the slowest. The EMIF control registers in the DSP allow the programmer to define SETUP, STROBE and HOLD lengths for the cycle that give a programmability to the access timing. For more control, the ARDY signal allows the logic designer to insert additional wait states to the STROBE timing as needed.

For the high speed data paths for A/D and DAC data in the Framework logic, burst accesses from the DSP provide the highest data rates. The EMIF configuration registers are set for SBSRAM memory interface timings, and in the logic the FIFOs respond to these signals to deliver data in continuous bursts. In the burst mode, one data point (64-bits wide) is provided for each clock. As can be seen from the read and write burst timing diagrams, data is at least two cycles latent from the control signals for reads, and may be zero for writes. On the '6416, a programmable latency allows the data to be up to 3 cycles latent for either reads or writes. The Framework Logic uses a latency of 3 for reads and 0 for writes.

Data bursts can be of any length and the Framework logic accommodates any burst length needed. Normally, this is set by the DMA channel.

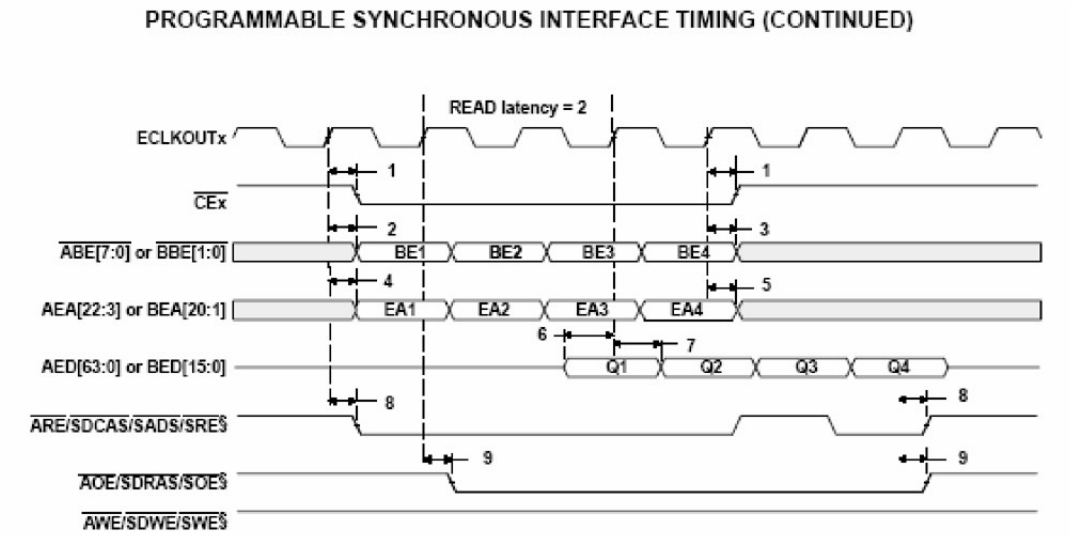


Figure 18. DSP Synchronous Read Timing. (Courtesy of Texas Instruments)

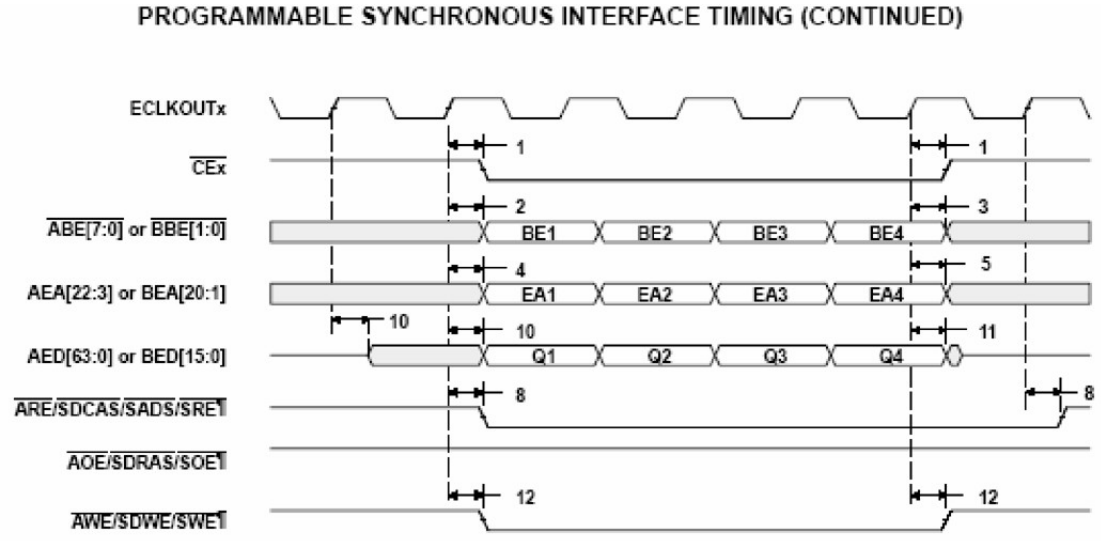


Figure 19. DSP Synchronous Write Timing. (Courtesy of Texas Instruments)

Since there are very few timing adjustments in the DSP EMIF control for sync registers, logic designers should be aware that burst interfaces require that the logic levels be minimized to meet timings. The Quixote logic has a simple data decoding and mux structure that allows the burst memories high speed, while penalizing slower async devices with a extra cycles for decoding and data delivery. Even then, only a small portion of the ACE0 memory space are read-back registers because of

the speed required. These read registers are grouped in the memory map so that a minimum of logic must be decoded and thereby maximizing the speed. Adding new read-back registers therefore should be done in a small memory region, requiring minimum logic.

The DSP memory is easily subdivided into memory types, such as async and burst by using different CE spaces. The DSP gives four CE signals for each EMIF that have timings as defined by the software (EMIF control registers) The memory map for the Framework Logic uses the following mappings.

```
component fifo_x64_fwft
-- converts standard Xilinx fifo to first word fall thru mode
   port (
        din:    IN  std_logic_VECTOR(63 downto 0);
        wr_en:  IN  std_logic;
        wr_clk: IN  std_logic;
        rd_en:  IN  std_logic;
        rd_clk: IN  std_logic;
        ainit:  IN  std_logic;
        dout:   OUT std_logic_VECTOR(63 downto 0);
        full:   OUT std_logic;
        empty:  OUT std_logic;
        wr_count: OUT std_logic_VECTOR(7 downto 0);
        rd_count: OUT std_logic_VECTOR(7 downto 0)
        );
end component;
```

Figure 20. Logic Component Source Code

Table 21. Dsp Memory Space Assignments

Memory Space	Type	Usage
EMIF A CE0	Async	Control and peripheral configuration registers.
EMIF A CE1	Burst	DAC and ADC FIFOs
EMIF A CE2	-	Not used
EMIF A CE3	-	Not used
EMIF B CE0	Burst	PCI FIFOs
EMIF B CE1	Async	PCI, timer and interrupt control registers.
EMIF B CE2	-	Not used
EMIF B CE3	Async	Test Register in Virtex2

DSP EMIF A Data Read Interface

The Framework logic is designed to support full data rate from the the high speed devices, with the slower speed devices subordinate. The following diagram shows an overview of the logic for the EMIF A data path.

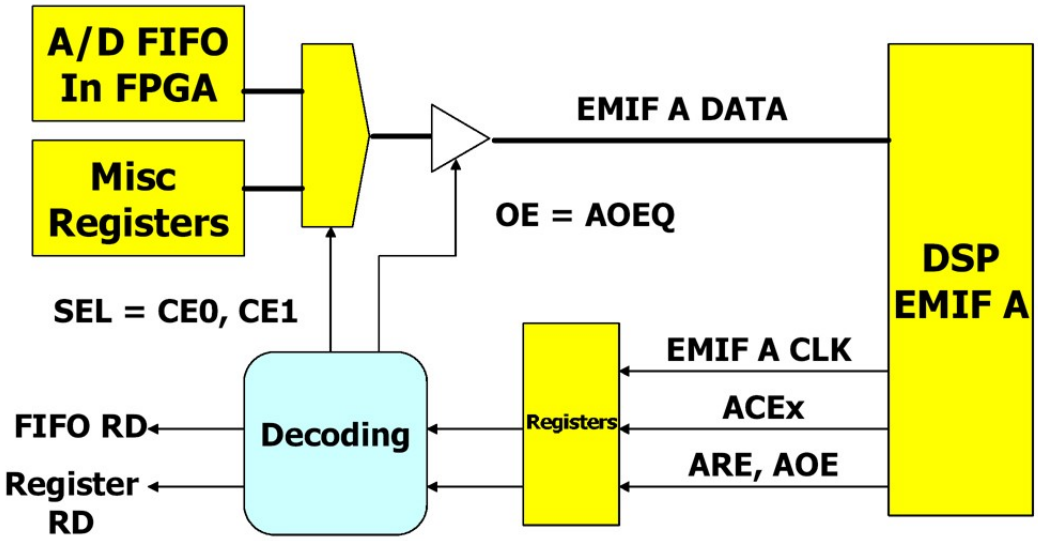


Figure 21. EMIF A Data Bus Read Interface Diagram

The first thing to notice in the diagram is that the control signals are all immediately registered as they enter the logic. This improves the timing by grabbing the signals at the IOB registers, and allowing the internal decoding to use the registered signals. A simple decoding of the memory spaces then enables the A/D data FIFO or the ASYNC_DATA sources. The async data sources have been selected from a large case statement, using minimal address decoding, that results in a registered data source (ASYNC_DATA) that is fed to the final data mux.

In the Framework Logic code, the async data mux is as shown.

```
-- async data sources
process (reset, aeclk, aea_q, ud_in, event_log_busy, event_fifo_not_empty, ad_fifo_status,
    dac_fifo_status, event_dout, pmc_j4_data)
begin
    if (reset = '1') then
       async_data_out <= (others => '0');
    elsif (rising_edge(aeclk)) then
       case aea_q(19 downto 17) is
           when "000" =>
               async_data_out <= X"000000" & ud_in;
--             async_data_out <= X"00000000" & test_word;
           when "001" =>
               async_data_out <= X"00000000" & sbsram0_read_data;
           when "010" =>
               async_data_out <= X"00000000" & sbsram1_read_data;
           when "011" =>
               async_data_out <= X"00000000" & pmc_io(63 downto 32);
           when "101" =>
               async_data_out <= X"00000000" & ad_fifo_status;
           when "110" =>
               async_data_out <= X"00000000" & dac_fifo_status;
           when "111" =>
               async_data_out <= X"00000000" & event_dout;
           when others =>
               async_data_out <= X"00000000" & Product_code & "0000" & Rev_Code &
                                 event_log_busy & event_fifo_not_empty & "00" & X"321";
       end case;
```

```
    end if;
end process;
```

As is shown, the many sources for data readback have been selected using only 3 address lines in the case statement, resulting minimal logic for the address decoding.

The final data mux picking between the A/D FIFO data or the async data is :

```
----------------------- dsp_data_out----------------------------------
-- the data mux
process (reset, aeclk, ad_fifo_d, async_data_out, ace_n_q)
begin
    if (reset = '1') then
        adata_out <= (others => '0');
    elsif (rising_edge(aeclk)) then
        if (ace_n_q(0) = '0') then
            adata_out <= async_data_out;
        else
            adata_out <= ad_fifo_d;
        end if;
    end if;
end process;
```

Notice that the data is selected simply on the basis of the registered EMIF A CE1 signal. Decoding for the read data is thus fast and simple.

As a last step, the data is enabled to drive the data bus for the reads. This is done using a simple tri-state output buffer (OBUFT)

```
aed <= adata_out when (adata_oe_n = '0') else (others => 'Z');
```

The output enable signal is a decode of the registered control signals

```
adata_oe_n <= '0' when
    ((ce0_n_q = '0' or ce1_n_q = '0' or (extend_ce1 = '1')) and aaoe_n = '0')
else '1';
```

Notice here that the output enable control was lengthened so that the data is output on the bus for the two cycles after CE goes away (refer to burst read timing diagram).

This DSP read interface has been successfully used in the Framework Logic to provide data to the DSP at burst rates using a 100 MHz EMIF A clock. The pipelined nature of the code allows the FPGA to meet timing and reduces many timing issues encountered because of the complex nature of the readback mechansim. The latency for the Framework Logic is 3 cycles, as set in the EMIF control registers.

DSP EMIF Write Interface

In general, writes are much easier than reads because we can pipeline the control signals and data to reduce the timing problems. In the Framework logic, the incoming data is immediately registered in the IOB pins to catch the data. In fact, a second register is used to align the data with the decoded signals used for registers and DAC FIFO writes.

```
----------------------- dsp_data_in----------------------------------
process (reset, aeclk)
```

```
begin
    if (reset = '1') then
        aed_q <= (others => '0');
        aed_qq <= (others => '0');
    elsif (rising_edge(aeclk)) then
        aed_q <= aed;
        aed_qq <= aed_q;
    end if;
end process;
```

Figure 22. EMIF A Incoming Data Bus Registers

Once the data has been captured, we can then write it into registers using decoded addresses. For example, this shows the control register for the analog trigger DAC.

```
-------- Analog trigger comparator control
process(reset, aeclk, anatrig_wr)
begin
    if(reset = '1') then
        tdac_clk <= '0';
        tdac_loadn <= '0';
        tdac_din <= '0';
    elsif(aeclk'event and aeclk = '1') then
        if(anatrig_wr = '1') then
            tdac_clk <= aed_qq(2);
            tdac_loadn <= aed_qq(1);
            tdac_din <= aed_qq(0);
        end if;
    end if;
end process;
```

EMIF B Interface

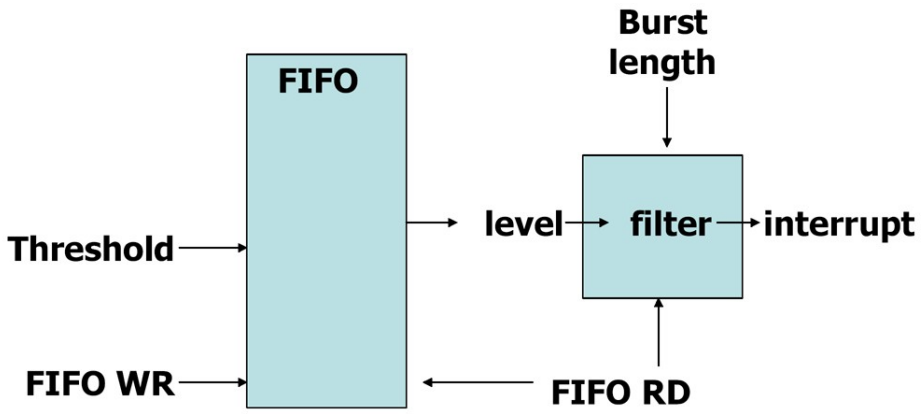
The EMIF B interface in the Framework Logic is primarily used as a pass-through to the PCI FIFOs. Data flowing to the PCI streaming engine in the Spartan2 logic passes through the Virtex2 with simple registers in the datapath to improve timing. A test register has been implemented to allow the DSP to verify its connection to the Virtex during test.

As can be seen from the memory map, two CE spaces onthe the EMIF B interface are open for use in the Framework Logic if the streaming engine interface is used. Interface methods similar to those used on EMIF A should be used since the bus is 100 MHz.

A/D and D/A FIFOs and Controls

FIFOs are used as the high speed data buffer to the DSP bus. The FIFOs may be read and written at the same time, allowing the DSP EMIF reads/writes to be independent of the data flow within the FPGA. Each FIFO also has code used to trigger an interrupt at the a programmable threshold level to allow either DMA or CPU to control the data flow.

Figure 23. FIFO Basic Interface



The threshold as shown is a programmable level where the FIFO can signal it requires more data from the DSP when used for the DACs, or must be read by the DSP when used for the A/Ds. During the data filling/draining operation, the level signal may transition through the threshold multiple times due to simultaneous reads and writes, so the interrupt is filtered so that it does not retrigger until a burst of data is read/written..

In the Framework Logic, both A/D and D/A data paths have a 512 element, 64 bit wide FIFO that holds the data. Also, keep in mind that each A/D and DAC have a 32K external FIFO, so these FIFOs internal to the logic are primarily for the DSP bus interface, not the bulk data buffering.

The FIFOs in the logic are implemented using Xilinx CoreLib functions. The FIFO component VHDL instantiation is shown here. This component wraps the Xilinx LogiCore async FIFO so that it has a first word fall through behavior by implementing a small state machine that pulls the first data point to the output register whenever it is put into the FIFO.

```
component fifo_x64_fwft
-- converts standard Xilinx fifo to first word fall thru mode
    port (
        din:       IN std_logic_VECTOR(63 downto 0);
        wr_en:     IN std_logic;
        wr_clk:    IN std_logic;
        rd_en:     IN std_logic;
        rd_clk:    IN std_logic;
        ainit:     IN std_logic;
        dout:     OUT std_logic_VECTOR(63 downto 0);
        full:     OUT std_logic;
        empty:    OUT std_logic;
        wr_count: OUT std_logic_VECTOR(7 downto 0);
        rd_count: OUT std_logic_VECTOR(7 downto 0)
    );
end component;
```

Figure 24. FIFO Component Instantiation

The FIFO interrupt filter, as contained in the fifo_util.vhd file, is shown in use here.

```
-- a/d fifo level filtering

ad_fifo_level_filter : fifo_level_flt
    Port map (
        rst       => reset,
        clk       => aeclk,
```

```
    fifo_en    => ad_fifo_rd_q,
    level_in   => adc_int_raw,
    level_out  => adc_int,
  active_lev   => vcc,
    burst_len  => ad_burst_length(6 downto 0),
    new_len    => ad_burst_wr,
    tp         => flt_tp );
```

Figure 25. Using The FIFO Interrupt Filter

This filter component takes in the threshold level interrupt as adc_int_raw and asserts it until the a_burst length is read from the FIFO. The interrupt given to the DSP DMA controller is adc_int. A counter in the code counts to the burst length using the ad_fifo_rd signal.

Decoding for the A/D read FIFO accesses is a simple register that is true when CE1 and ARE are true.

```
process (reset, aeclk, ce1_n_q, aare_n_q)
begin
    if (reset='1') then
        ace1_burst_rd <= '0';
    elsif (rising_edge(aeclk)) then
        if (ce1_n_q = '0' and aare_n_q = '0') then
            ace1_burst_rd <= '1';
        else
            ace1_burst_rd <= '0';
        end if;
    end if;
end process;
```

Figure 26. A/D FIFO Read Control Signal

Notice that the control signal decoding uses the registered control signals for improved timing.

The DAC FIFO is handled in a similar fashion, with writes to the FIFO being controlled as such

```
-- DSP burst write
process (reset, aeclk, ce1_n_q, aawe_n_q)
begin
   if (reset = '1') then
       ace1_burst_wr <= '0';
   elsif (rising_edge(aeclk)) then
       if (ce1_n_q = '0' and aawe_n_q = '0') then
           ace1_burst_wr <= '1';
       else
           ace1_burst_wr <= '0';
       end if;
    end if;
end process;
```

Figure 27. DAC FIFO Control Signal

SBSRAM Memory

The Quixote has two banks of 1M x 32 ZBT SBSRAM (synchronous burst, Zero Bus Turn-around SRAM) memory comprised of four Cypress Technology CY7C1372BV25-167AC (or equivalent) chips. These devices have a flow-through architecture and are 2.5V IO. The SBSRAM supports up to 167 MHz (~600 Mbyte/sec) data rates during burst accesses. This memory may be used by the FPGA in any process as required by the application and has completely independently controls. Since this SBSRAM provides the highest data rates only in burst mode, it is suggested that the user arrange data to take advantage of this architecture.

The sample framework code provided for the FPGA is derived from Xilinx XAPP163. The application note has a detailed description of the interface and its use. In the framework logic, a simple SBSRAM interface is shown that merely demonstrates that the RAM is connected and working. It consists of two registers: address and data. To use, write an address to the SBSRAM address register, then read or write from that address. More complex control is left to the application as required.

Digital I/O from the Virtex2

There are 40 bits of digital I/O from the Virtex to the MDR50 connector. These are configured as a simple LVTTL register, with software-programmed direction on a nibble basis, in the Framework logic. A number of logic standards can be programmed into the Virtex for high speed use such as LVDS and low voltage TTL as constrained by the Virtex2 chip pins. The MDR cable is arranged such that the logic is paired on the cable ()-1, 2-3, ...39-40) for differential use. The pad numbers for the assignments are shown in the UCF file as UD<x>, and at the hardware appendix of this document.

These pins are NOT 5V compatible! Protection devices on each pin limit the voltage to 3.3V or less to prevent damage to the Virtex2 from overvoltage or electrostatic damage. Input signals above 3.3V will be clamped to 3.3V and can sink ~1A of current for short periods. Do not use this as a signal limiting mechanism however since it is meant for protection only.

Clock Domains

Judicious choice of clock domain boundaries, and careful handling of any transition across the clock boundaries is crucial to a reliable design. Past experience has shown that more problems occur on this topic than any other. In the Quixote design, the EMIF A clock is a 100 MHz fixed rate, with no phase relationship to either the DAC or A/D sample clocks. In this case, a FIFO is used as the clock domain transition for the main data path because of the ease of use and reliability. Control registers are also in the EMIF clock domain, since most are static controls. In the case of non-static controls, signals should handshake across the clock domain boundary. Xilinx has an excellent application note on this topic that is worth review for general instruction in this topic.

The following diagram shows the clock domains for the Framework Logic. Each clock domain transition has been chosen to occur at FIFO boundaries since the FIFOs support asynchronous clocks. This greatly simplifies the clock domain transitions in a clean way.

As can be seen from the clock domain diagram, the EMIF B clock is used solely for the EMIF B pass-through. While it may be the same frequency as EMIF A, there is no guarantee that the two clocks are in-phase.

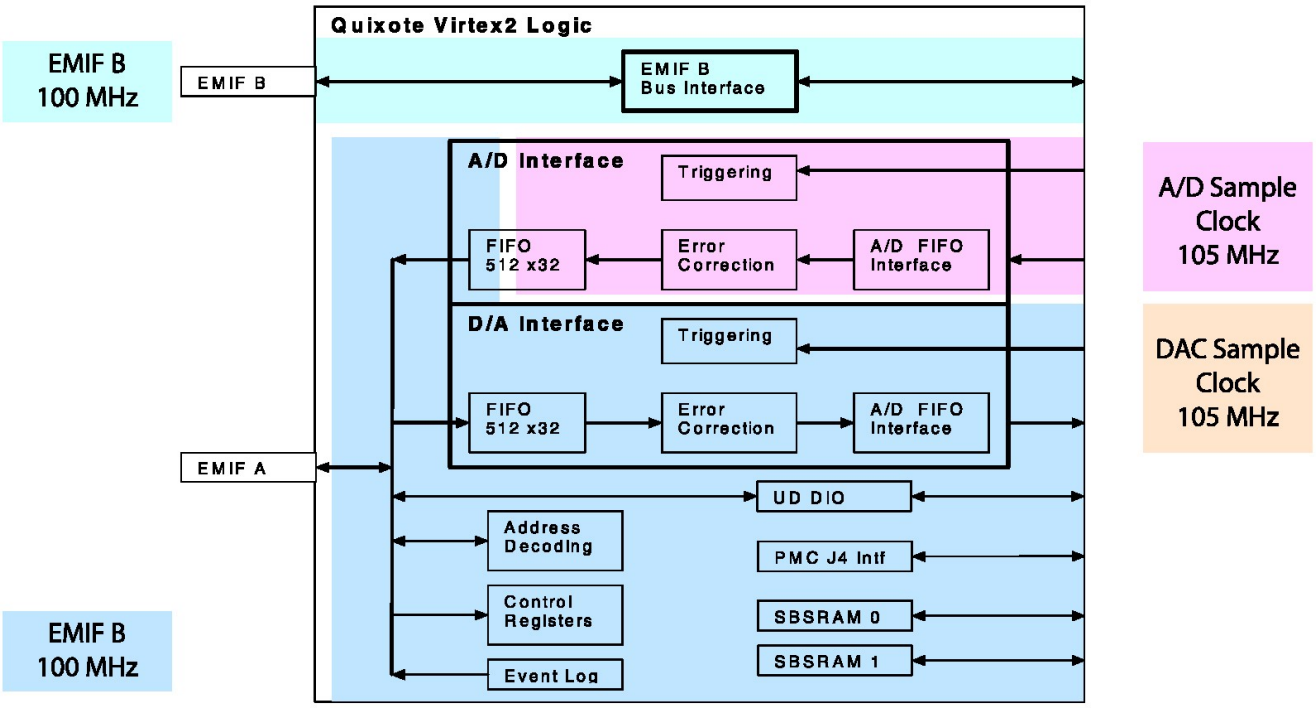


Figure 28. Clock Domains in The Framework Logic

Clocks for Use in the Logic

There are several clocks available to the designer in the logic that are intended for different functions: as shown in the following table.

Table 22. Available Clocks for The Logic

Clock	Function	Frequency	Pin (FF1152 package 6M device)	Pin (FF896 package 2M device)
aeclkout1	DSP EMIF A clock	~100 MHz	AE18	AD16
beclk	DSP EMIF B clock	~100 MHz	AG17	AE15
refclk (dds_src_clk)	Reference clock source	20.00 MHz, 5 ppm	AK17	AH15
ADC_CLK	clock from Spartan2 clock matrix	varies, nominally 0-100 MHz	E16	C14
DAC_CLK	clock from Spartan2 clock matrix	varies, nominally 0-100 MHz	H16	F14
PLL clock (differentially received)	Software programmable sample clock with <6 ps jitter	programmable from 50-105 MHz	E19 (+) E18 (-)	C17 (+) C16 (-)

The DSP EMIF clocks should be used for synchronizing to the DSP EMIF A or B buses. Since these clocks are fixed frequency at ~100 MHz, a Digital Clock Manager (DCM) inside the Virtex2 should be used to reduce the clock skew in the chip, as is shown in the Framework Logic. These clocks are not controlled for jitter or absolute frequency and should not be used for precision timing operations or analog sample clocks.

The 20 MHz reference clock is a precision 20 MHz with about 5ppm of drift that may also be accurately calibrated if necessary to use as a stable, accurate timebase. This clock is from a temperature compensated crystal oscillator (TCXO) (Vectron P/N VTA2-1B0-20.0MHz) and is very stable over time.

The PLL clock is normally used as the sample clock for the A/D and D/A. This is a software programmable frequency source with very low jitter (<6 ps jitter) and resolutions of 200kHz. This allows for multiple band sampling of IF in may RF applications. Since this clock must be low jitter to preserve the analog signal quality, it is important in the logic design to preserve this quality of clock. Do NOT use a DCM to manipulate this clock internally before sending to the analog converters, since a DCM has large jitter of over 100 ps and your signal quality will be lost. If you must divide the clock down, use a simple flip-flop divider for cleanest results. The PLL clock is received as a differential PECL signal from the PLL directly into the Virtex2 logic and given to the Spartan2 as LVTTL for use in the timing selection matrix.

The Quixote also supports clocks to be supplied from external sources such as PXI or as inputs to the card. This supports multi-card synchronization and the use of custom external timebases. These clocks are steered by the Spartan2 into the Virtex2 through a clock selection matrix. The software library in Pismo supports clock steering and selection to two pins on the Virtex2 : ADCLK and DACLK. These two clock sources may then be used in the Virtex2 for sampling and timing.

Pin Banks

The logic has 8 pin banks that must use a compatible IO standard for all pins on that bank. Quixote has two banks, 0 and 1, that are LCVCMOS 2.5V for the SBSRAM, and the other six banks (2-7) that are 3.3V LVTTL, LVDS or a few other standards. These bank assignments cannot change, so pin re-mappings and IO standards must follow this constraint.

PMC J4 Test Port

The framework example logic contains a simple port at 0x80560000 that is used to test the connections to the PMC J4 connector in production test. This test port is implemented as a 32 bit output port for PMC J4 pins 1-32, and as an input port for pins 32-64. In the test procedure, the signals are connected through a test fixture as 1 to 33, 2 to 34, ..., 32 to 64. Patterns are then written to the port to verify that all pins are connected to the logic and that no pins are shorted together.

This test implementation can be easily modified to allow custom developments to use the J4 connector. The memory mapped location used for the test port can then be remapped to the requirements of the application.

Constraints

There are several important classes of constraints used by the Framework Logic : timing, pin placement and IO standards. These constraints are shown in the .ucf (user constraint file) that is used during the fitting process.

The constraint files are different for the QuixoteII (2M gates Virtex2) and Quxitoe III (6M gates Virtex2) devices since they are in different packages. The 2M part is in a FF896 package; the 6M part is in a FF1152 package. The respective constraint files are quix_intf_ff896.ucf and quix_intf_ff1152.ucf.

Timing Constraints

The timing constraints defined cover the clocks used in the design and the external device signal timing. Clock period period constraints are used to cover most of the logic since they define the clock rate for all flip-flops connected to that clock. These period constraints then cover most of the logic paths used in a synchronous design. Here are the clock period constraints used by the Framework Logic:

```
TIMESPEC "TS_aeclkout1"=PERIOD "aeclkout1" 9.8 ns ; # was 9.6 ns DPM 8/7/03
TIMESPEC "TS_adc_clk" = PERIOD "adc_clk" 9.95 ns ; # created this so multicycle
   spec will work
NET "beclkout1" PERIOD = 9.6 ns ;
NET "adc0_clk" PERIOD = 10 ns ;
NET "adc0_clk_n" PERIOD = 10 ns ;
NET "adc0_wclk" PERIOD = 10 ns ;
NET "adc0_rclk" PERIOD = 7 ns ;
NET "adc1_clk" PERIOD = 10 ns ;
NET "adc1_clk_n" PERIOD = 10 ns ;
NET "adc1_wclk" PERIOD = 10 ns ;
NET "adc1_rclk" PERIOD = 7 ns ;
NET "dac0_wclk" PERIOD = 7 ns ;
NET "dac0_rclk" PERIOD = 10 ns ;
NET "dac1_wclk" PERIOD = 7 ns ;
NET "dac1_rclk" PERIOD = 10 ns ;
NET "adc_clk" PERIOD = 10 ns ;
NET "dac_clk" PERIOD = 10 ns ;
```

Figure 29. Clock Period Constraints

As can be seen, EMIF A clock (aeclkout1) is constrained to 9.8 ns, giving a small margin for a 100 MHz bus. EMIF B (beclkout1) is similarly constrained to 9.8 ns. Other clocks include the highest A./D and DAC clock rates.

External devices require an additional constraint to be sure that we get the signal on-chip and to its destination in time. Since the external chip, such as the DSP, may have a delay from the clock edge to when we get the signal, an additional constraint is defines the amount of time after the clock that the signal is given to the logic. This type of constraint is used on the DSP control signals such as CE, ARE, AWE and addresses to guarantee that setup timings are met. a Timing Group is defined for these signals with a timing constraint for the group.

```
TIMESPEC "TS_DSP_FAST_CKIN" = FROM "DSP_FAST_CKIN" TO "FFS" 4;

NET "ace_n<1>" TNM =DSP_FAST_CKIN ;
NET "ace_n<0>" TNM =DSP_FAST_CKIN ;
NET "aawe_n" TNM =DSP_FAST_CKIN ;
NET "aare_n" TNM =DSP_FAST_CKIN ;
NET "aaoe_n" TNM =DSP_FAST_CKIN ;
NET "aea<22>" TNM =DSP_FAST_CKIN ;
NET "aea<21>" TNM =DSP_FAST_CKIN ;
NET "aea<20>" TNM =DSP_FAST_CKIN ;
NET "aea<19>" TNM =DSP_FAST_CKIN ;
NET "aea<18>" TNM =DSP_FAST_CKIN ;
NET "aea<17>" TNM =DSP_FAST_CKIN ;
NET "aea<16>" TNM =DSP_FAST_CKIN ;
```

Figure 30. DSP Signals Timing Constraint

The DSP signals have 4 ns worst case timing from clock to signal valid that is accounted for in this consraint.

IOB Constraints

It is also important to capture many of the high speed signals in the IOB registers for timing. THis is done using a constraint as in

```
# A/D data pipeline input latches in IOBs
INST "quixote_ad_adc0_pipe_adc_din_q_0" IOB = TRUE;
```

Figure 31. An IOB Register Constraint

IO Standard Constraints

Each pin is defined for its appropriate IO standard. The XIlinx default standard is LVTTL, so it is common to leave those out and define only the exceptions. A notable exception is the differential PECL clock output to the A/D as is shown here. Do not change the IO standard as defined as this may cause damage to the devices.

```
N E T " a d c 0 _ c l k " I O S T A N D A R D = L V P E C L _ 3 3 ;
```

Figure 32. IO Standard Constraint

Pin Constraints

Each pin has a placement defined in the UCF file, as required by the circuit board design. DO NOT CHANGE these assignments as damage may occur to the Quixote! They must be used on all compiles.

```
NET "adc0_clk_n" LOC = "T11" ;
```

Figure 33. A Pin Sample Pin Constraint

Simulation

The test files are used in the simulation and testing of the Framework code. The testbench file is TB_quixote.vhd and it uses several components for testing that are defined by the other model files. These model files are very simple and are only for simple testing only. More complex models may be needed to adequately model more advanced uses.

The testbench contains a set of simulation steps that exercise various functions on the framework logic for basic interface testing. Behavioral procedures have been written to simulate the DSP timing for sync and async memory accesses that are useful in simulating data movement. Also, the steps to setup the logic for data streaming support are shown so that interrupt servicing (DMA or CPU accesses), trigger and event log use are illustrated.

Required Software and Hardware

Recommended Software: Mentor Graphics ModelSim 5.7e or higher

Recommended Hardware: Pentium 3 or better at 500 MHz with 512M of RAM

Setting Up the Simulation

The files unzipped from the Framework Logic archive contain all the source and macro files needed. You will normally need to make a ModelSim project reflecting your exact directory structure, although an mpf file is provided. You will also need to compile the Xilinx unisim, simprim and core libs and point to them in ModelSim. Set the project default to VHDL '93 if you intend to compile in ModelSim using the project manager.

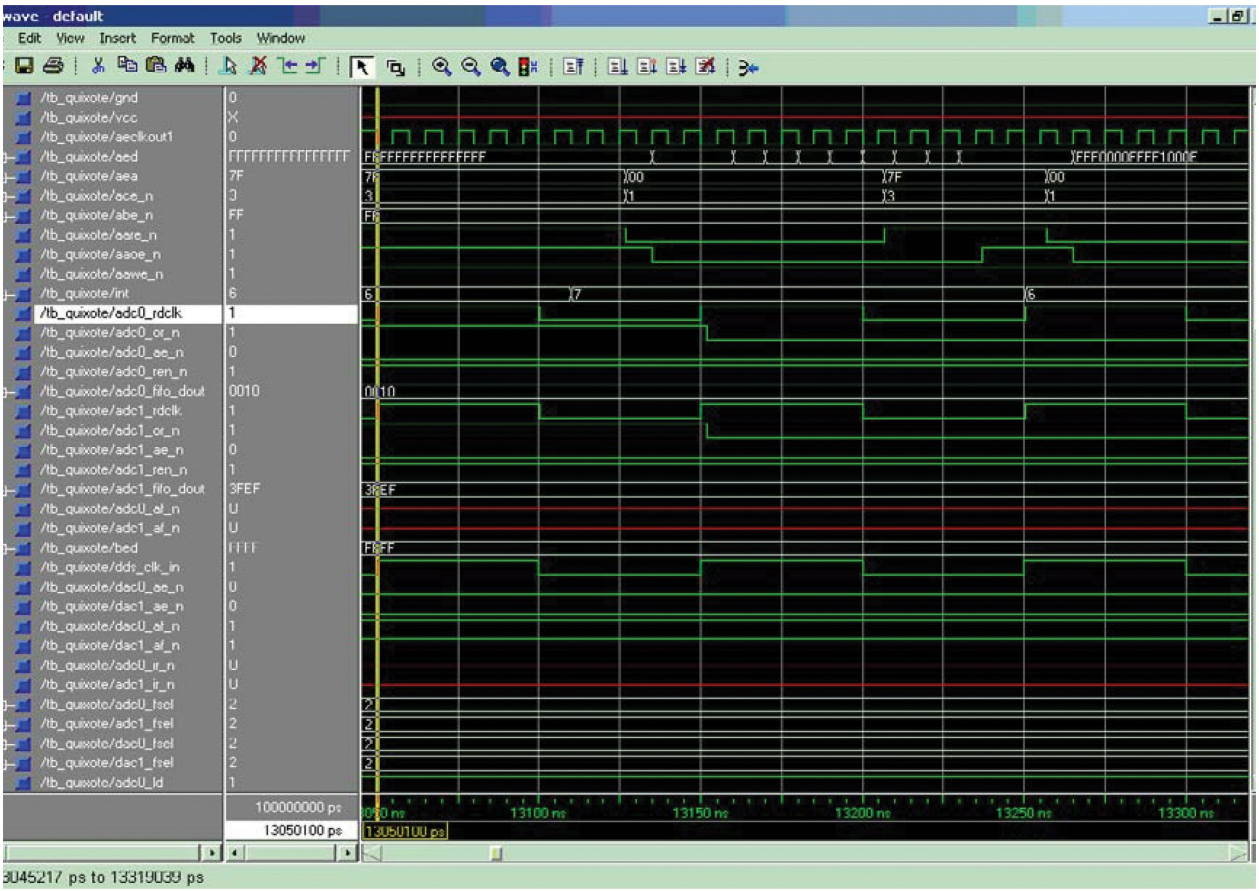
Loading the Testbench

The simulator used from within ISE is Mentor Graphics ModelSim and support files for using the testbench from within environment are included. We recommend that you run ModelSim using the macro files provided as a standalone operation, and not from within ISE. This is because we have several models that do not get compiled if you run from ISE.

The macro files are the tb_quixote.do and wave.do files for compilation and waveform visualization, respectively. The tb_quixote.do file compiles the files in the necessary order and loads the simulation.

Running the Simulations

Once you have executed the macro files in ModelSim, the simulation is ready to run. A time period of 100 us is usually enough to see it start. Here is a sample of the output showing a DMA burst from the A/ D FIFO to the DSP.



Modifying the Simulations

The testbench file provides sample code to begin your simulations. Accesses are shown to each peripheral from the DSP, and how to control the peripherals such as A/D and DACs. In the main process, the DSP enables the A/Ds to take data and then services interrupts from the FIFOs. DAC accesses are also shown. Internal signals for the data streams can be viewed as well.

In the code are several procedures that simulate DSP accesses and can be used to simulate both async and burst DSP access to the logic. The following code shows the DSP async access procedure, with the correct timing, and its use in the testbench.

```
-- writes to 'C6414 ACE0, modeling async accesses
procedure write_ACE0 (
    constant address : bit_vector(22 downto 16);
    constant data:     bit_vector(63 downto 0)
) is
    begin
        wait until aeclkout1'event and aeclkout1 = '1';
        ace_n(0) <= '0' after 2 ns;
        aea <= to_stdlogicvector(address) after 2 ns;
        aed <= to_stdlogicvector(data) after 2 ns;
        aaoe_n <= '1';
        wait until aeclkout1'event and aeclkout1 = '1';  -- 1 clock of setup time
        aawe_n <= '0' after 5 ns;                        -- worst case timing
        wait until aeclkout1'event and aeclkout1 = '1';  -- 2 clocks of strobe time
        wait until aeclkout1'event and aeclkout1 = '1';
        aawe_n <= '1' after 5 ns; -- worst case timing
```

```
    wait until aeclkout1'event and aeclkout1 = '1'; -- 1 clock of hold time
    ace_n(0) <= '1' after 2 ns;
    aea <= (others => 'H') after 2 ns;
    aed <= (others => 'H') after 2 ns;
    aaoe_n <= '1';
end write_ACE0;
```

Figure 34. VHDL Procedure for Simulating a DSP Async Access

```
procedure write_ACE0 (
    constant address : bit_vector(22 downto 16);
    constant data: bit_vector(63 downto 0) )

procedure read_ACE0 (
    constant address : bit_vector(22 downto 16);
    constant data: bit_vector(63 downto 0) )

In use…..

--- write to UD
write_ACE0("0110000", X"00000000000000FF"); -- ud config
write_ACE0("1010000", X"0000000A98765432"); -- ud data
gap;
read_ACE0("1010000");
```

Figure 35. Simulating a DSP Async Access

There are also testbench procedures for the burst accesses

```
-- reads from 'C6414 ACE1, modeling FIFO-style burst sync accesses (burst length = 8)
procedure read_ACE1 (
    constant address : bit_vector(22 downto 16)
) is
    begin
        wait until aeclkout1'event and aeclkout1 = '1';
        ace_n <= "01" after 2 ns;
        aea <= to_stdlogicvector(address) after 2 ns;
        aed <= (others => 'H');
        aare_n <= '0' after 2 ns; -- variable by external clock phase correction
        wait until aeclkout1'event and aeclkout1 = '1'; -- 1 clock before aaoe_n (data word 0)
        aaoe_n <= '0';
        wait until aeclkout1'event and aeclkout1 = '1'; -- 1 clock of strobe time (data word 1)
        wait until aeclkout1'event and aeclkout1 = '1'; -- 1 clock of strobe time (data word 2)
        wait until aeclkout1'event and aeclkout1 = '1'; -- 1 clock of strobe time (data word 3)
        wait until aeclkout1'event and aeclkout1 = '1'; -- 1 clock of strobe time (data word 4)
        wait until aeclkout1'event and aeclkout1 = '1'; -- 1 clock of strobe time (data word 5)
        wait until aeclkout1'event and aeclkout1 = '1'; -- 1 clock of strobe time (data word 6)
        wait until aeclkout1'event and aeclkout1 = '1'; -- 1 clock of strobe time (data word 7)

        ace_n <= "11" after 2 ns;
        aare_n <= '1' after 2 ns;
        aea <= (others => 'H') after 2 ns;
        wait until aeclkout1'event and aeclkout1 = '1'; -- 3 clocks of latency
        wait until aeclkout1'event and aeclkout1 = '1';
        wait until aeclkout1'event and aeclkout1 = '1';

        aaoe_n <= '1' after 2 ns;

        wait until aeclkout1'event and aeclkout1 = '1'; -- gap time
        end read_ACE1;
```

Figure 36. VHDL Procedure for DSP Burst Read Access

This sample code simulates a burst read of 8 64-bit data words. Other burst lengths can be done by simply modifying the code for the length of burst required.

Finally, it is also possible to simulate the DSP interrupt service routines in the testbench using the LOOP_INT procedure.

```
procedure LOOP_INT (
    constant interrupt : in integer range 0 to 3;
    constant ADDRESS : in bit_vector(22 downto 16);
    constant BLKSIZE : in integer range 0 to 511
    )

procedure LOOP_DAC_INT (
    constant interrupt : in integer range 0 to 3;
    constant ADDRESS : in bit_vector(22 downto 16);
    constant BLKSIZE : in integer range 0 to 511;
    constant data : in bit_vector(63 downto 0)
    )

In use….

for i in 0 to 2 loop -- service 3 interrupts
    LOOP_INT(0,"0000000",24); -- services int0 = A/D
end loop;

-- prefill
for i in 0 to 3 loop
    LOOP_DAC_INT(1,"0000000",8, X"0302FFF90300FFF8");
end loop;
```

Figure 37. Simulating DSP Interrupt Servicing

Some Things to Watch Out For

- Be sure to use a time resolution of ps for all simulations. The DCM simulation component will not work reliably unless ps resolution is used.
- Many of the files use VHDL ‘93 and will issue errors if not compiled as ‘93 code. As a rule, always use ‘93 standard when compiling the Framework Logic.
- The DCM reset code in quixote_intf.vhd keeps the DCM in reset for ~205 us (4096 reference clocks) to meet the Xilinx DCM reset requirement. This is required for all real compilations and must be included. For simulation work however, this reset period is quite long and can be shorted to save time. A line of code is provided in the source for both timing in the quixote_intf.vhd file (search on dcm_rst).

```
-- DCM reset generator - conditions reset per Virtex2 DCM requirements
process (reset, ref_clk, a_count)
begin
    if (reset = '1') then
        dcm_rst_state <= idle;
    elsif (rising_edge(ref_clk)) then
        case dcm_rst_state is
```

```
        when idle =>
            if(a_count(12) = '1') then -- when board comes out of reset and time delay expired
--          if(a_count(3) = '1') then -- test version, faster for easier simulation
                dcm_rst_state <= wait1; -- and both clock inputs present
            else
                dcm_rst_state <= idle;
            end if;
```

Figure 38. DCM Reset Code Fragment

Another problem encountered on Quixote designs can be when the logic is sparsely used. The 2M and 6M devices are big chips, and as such can have long paths internally when not much is in there. Adding a few pipeline registers to break up the long paths can be required to meet timing when the logic is sparse.

The A/D minimum sample rate is 20 MHz. AC performance is seriously compromised at slow rates and it is recommended that the A/Ds be run at the highest possible sample rate. A simple decimator in the logic is generally used for this and can be easily implemented to allow the A/Ds to run at high rates even if the data stream does not need the data that fast.

Loading the Logic Image

For the Quixote, usually an EXORMacs format text file must be generated from the output .BIT file produced by the place and route process. This is done by opening the PROM File Formatter utility from within the ISE and converting the BIT file into an EXO file. The PROM properties must be set to "byte- wide" and EXORmacs to generate the required EXO file. Once the properties are set, the .EXO file is generated in the source directory.

To load the new .EXO file into Virtex2, use the logic downloading applet included in the software release. Within the program, read in the EXO file you wish to use from the host disk, and program the Virtex2. Since the logic is RAM based, it must be loaded each time the card is powered up. Resets to the card do not require a logic reload. The programming process depends on the size of the design being loaded, but typically takes less than 10 seconds. Wait until it finishes before doing anything else.

The logic loading applet features both a Windows user interface and a command-line interface. You may add a link to the applet within the Windows startup folder specifying the name of an EXO file to automatically load immediately after Windows boots. See the Applet chapter for more details on the logic loader applet program.

Alternately, the Malibu library baseboard object provides a number of features to to support downloading of EXO logic images directly from within your Windows application code. These features are useful in situations where the configuration of the baseboard logic must be changed "on-the fly" -- during the execution of a Host and/or DSP program. See the chapters on the baseboard software for details on loading the logic in an application.

Quixote Analog Input and Output

Analog Inputs

Overview

Quixote implements dual channels of analog input that allow simultaneous sampling at 105 MHz with 14-bit resolution. Each channel is an independent signal chain with no signal multiplexing, allowing for true simultaneous sampling and better channel isolation. Both channels must run synchronously. A wide variety of trigger mechanisms have been provided.

All channels have 50 ohm inputs with a standard input range of +/-1V. A six-pole anti-alias filter precedes the A/D converter. Custom ranges and filters may be special ordered, contact Innovative Integration sales department.

Data collection is controlled by the triggering mechanisms which allow you to control when data is captured. Triggering is defined as the time period for data collection bounded by a start trigger and stop trigger. During the active trigger period, the timebase defines when the samples are acquired. Various sources for start trigger, stop trigger, and the timebase including external inputs allow a variety of data collection methods to be used.

The analog inputs are digitally corrected for gain and offset errors in the FPGA using a first order model. The error correction is done real-time as each data point is collected. Calibration coefficients are determined either at factory calibration or derived from the auto-calibration of the card are used in the error correction. The coefficients are stored in an on-card non-volatile memory and must be loaded into the error correction logic before use. This is further discussed in the calibration section.

The A/D points from the enabled channels are held in a 32K sample FIFO for consumption by the FPGA. In the standard FPGA logic, the samples are error corrected and moved to second FIFO in the FPGA for DSP access. These points are presented to the DSP as four 14-bit samples stacked on the 64- bit A bus. Each 64-bit number has two pairs of two's-complement numbers stacked with channel 0 oldest sample on bytes 1-0, channel 1 on bytes 3-2, and the newer sample pair stacked on bytes 5-4 for channel 0 and 7-6 for channel 1. These points can be consumed by the DSP as DMA or CPU reads, using programmable level interrupts from the FIFO.

The following block diagram shows the functional interconnections of the analog input subsystem.

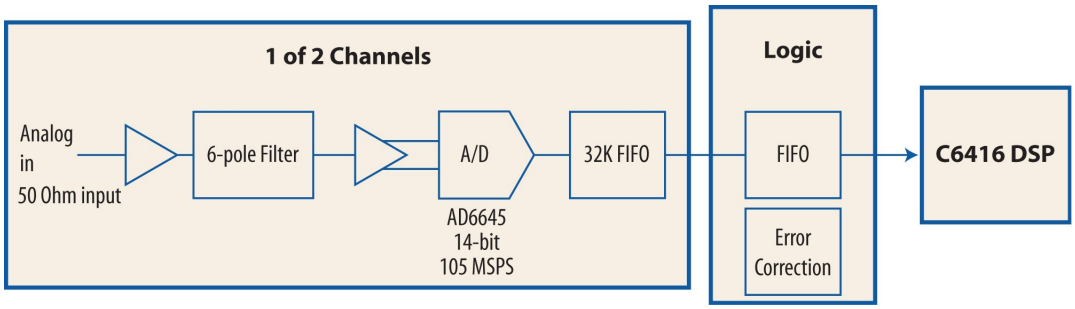


Figure 39. Interconnections of the Analog Input Sub-System

A/D Anti-alias Filtering and Input Circuitry

The analog input circuitry consists of an SMB coaxial input into a 50 ohm input amplifier followed by an analog multipole anti-alias filter. This input circuitry conditions the signal for the A/D input range and removes out-of-band noise. The 50 ohm input is intended to match the impedance of the input cables to minimize signal reflections and loss.

Analog signal input range is +/-1V on the standard Quixote. Custom input ranges are available, contact Innovative Integration sales department for custom ordering details.

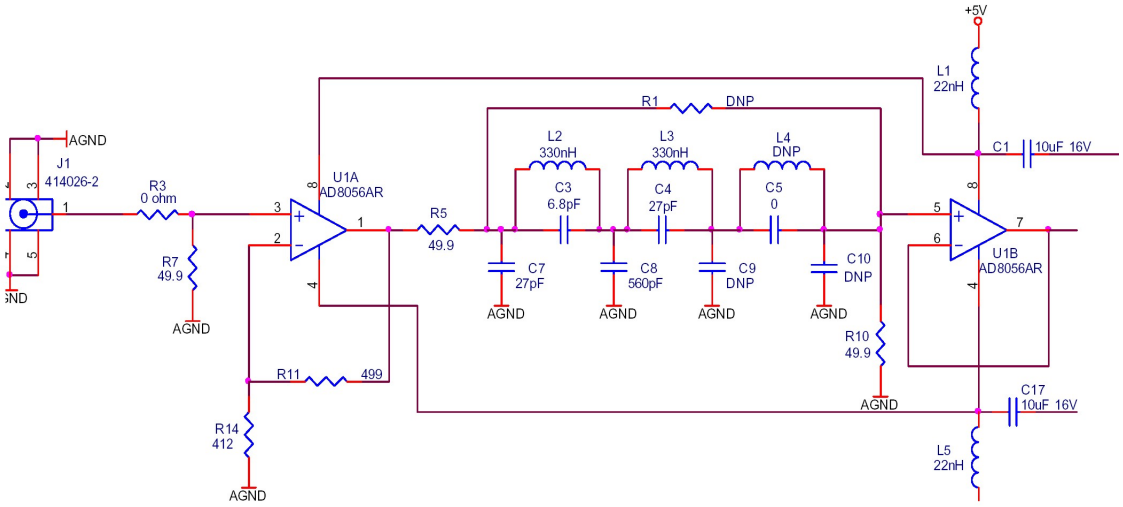


Figure 40. Input Filter Circuitry

The input anti-alias filter is a continuous-time analog filter. The filter is a multipole design, with the nominal -3dB point at 13.3 MHz. The following two graphs show the filter frequency response characteristics for the 100 kHz to 100 MHz frequency range, and the detailed 1 MHz - 50 MHz range. Custom filters can be special ordered from Innovative Integration to suit your application, contact our sales department.

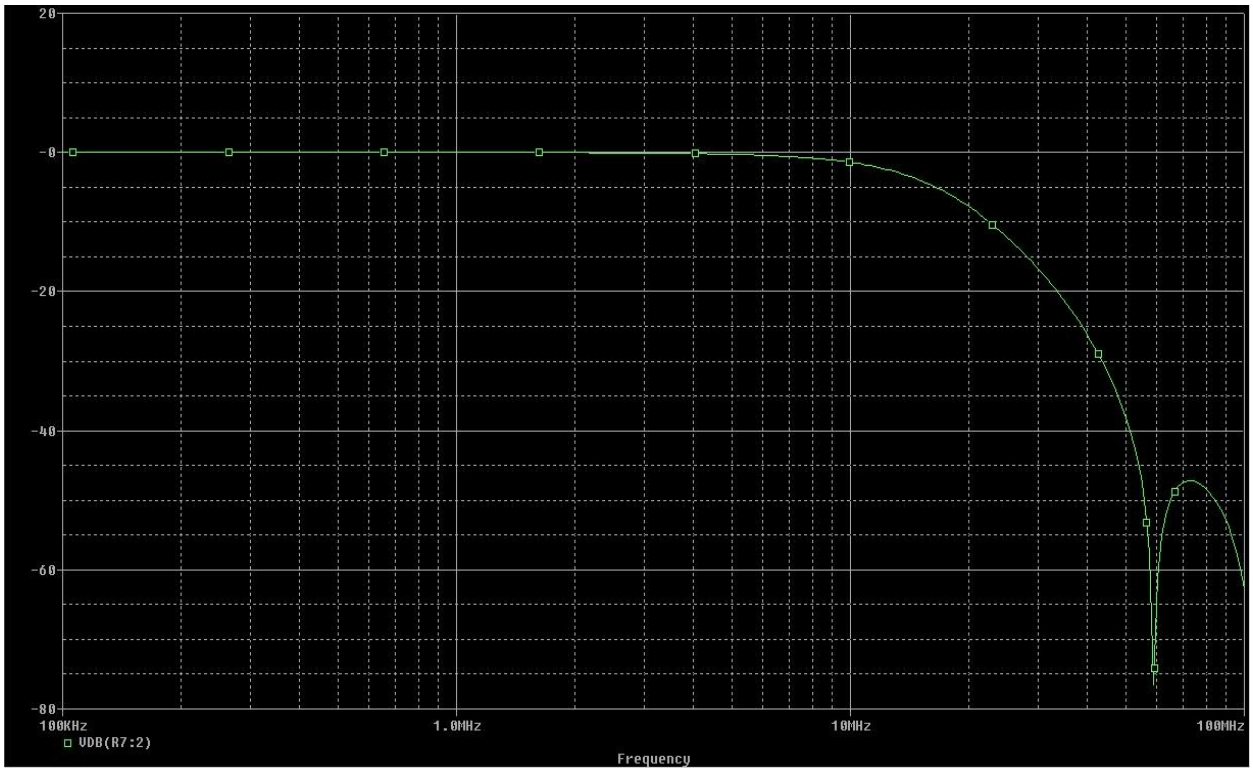


Figure 41. A/D anti-alias filter frequency response

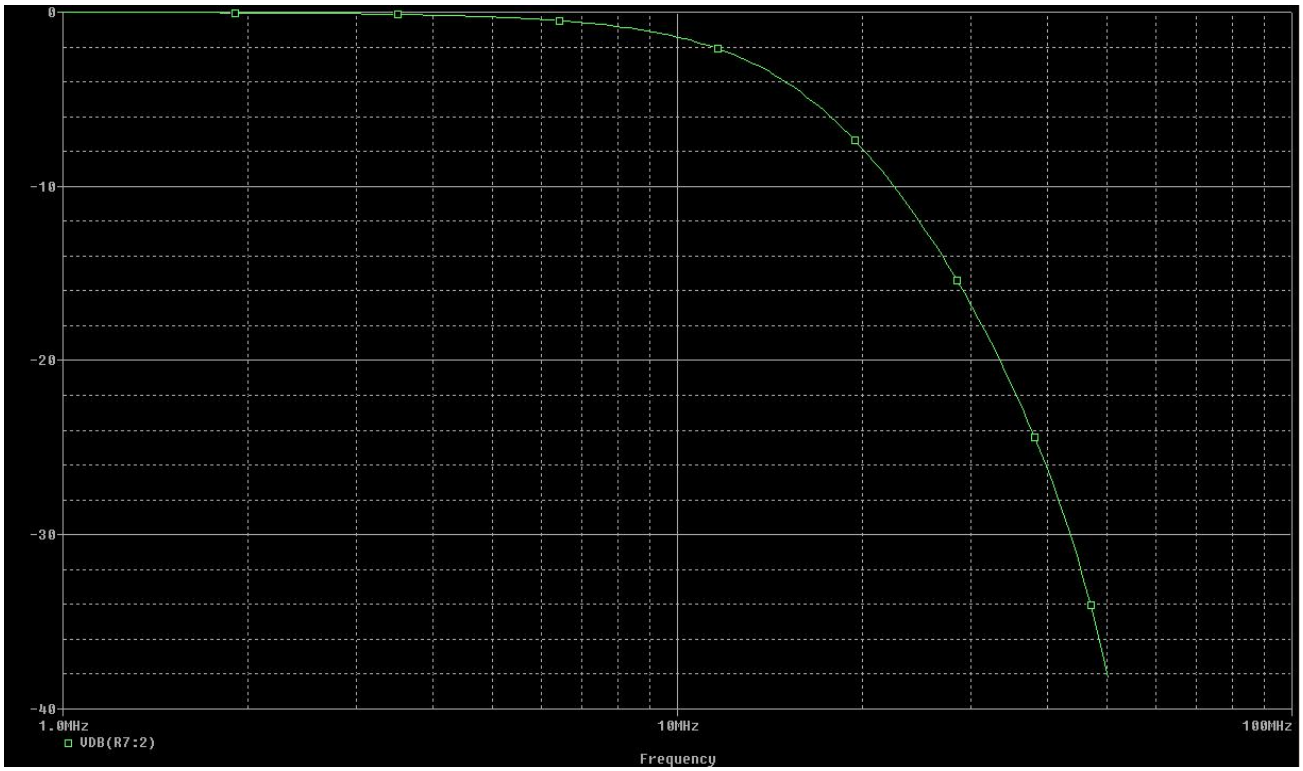


Figure 42. Detail of A/D anti-alias filter frequency response

A/D Triggering and Data Collection

The A/D data acquisition engine in the standard FPGA logic has the primary task of collecting the data from the A/D's based on the triggering conditions set by the DSP. This allows the programmer to acquire data at very precise times, under specified conditions, resulting in a data set comprised only of the data of interest. The DSP controls the data acquisition process by enabling the data acquisition, initializing triggering parameters, and moving the data from the external sample FIFO, through the error correction logic, to the internal DSP bus FIFO after collection.

Enabling the data acquisition engine

The data acquisition engine is enabled by the A/D RUN bit in the analog control register (0x80320000, bit 0). The A/D RUN bit, when set to '1', enables the data collection engine to take data when the triggering conditions are satisfied. The A/D RUN bit must be true for triggers to be recognized, for the data to flow, and essentially acts as a reset to the A/D data collection process and triggering mechanism. Triggers may be cleared and the data engine reset by making A/D RUN false. The fifo must be cleared separately by the A/D FIFO RESET bit (0x80000000, bit 13).

Data Format

Data is always stored as sets in ascending channel order, with the newest data pair on the upper half of the 64-bit output word. Figure 19 shows the data word format and the data set accumulating over time in the FIFO. All data is two's-complement numbers 16-bit numbers, with 14 bit precision. Samples taken at time t comprise the first data set, with the channels ordered as shown, samples taken at time t+1 comprise the second data set. This format continues *ad infinitum*.

Table 23. A/D channel ordering and data set format

Sample time	Byte 7-6	Byte 5-4	Byte 3-2	Byte 1-0
t+1, t	channel 1 @ t+1	channel 0 @ t+1	channel 1 @ t	channel 0 @ t
t+3, t+2	channel 1 @ t+3	channel 0 @ t+3	channel 1 @ t+2	channel 0 @ t+2
t+(n+1), t+n	channel 1 @ t+(n+1)	channel 0 @ t+(n+1)	channel 1 @ t+n	channel 0 @ t+n

Since the DSP interacts primarily with the FIFO, which has a very high speed burst memory interface capable of 1GB/sec bursts, Quixote is able to support high data rates with minimal bus loading - leaving more CPU time for computation. In order to optimize data transfer rates from the A/D FIFO, A/D data is read from the FIFO using synchronous access cycles on the 'C6416 EMIF A interface. This allows one word, comprised of 2 data samples per channel, to be read from the FIFO per EMIF clock cycle in burst transactions after a 2 clock setup per burst transfer. The DSP external memory interface must be initialized as a burst RAM (SBSRAM) memory space for proper operation. Quixote's Pismo Toolset software automatically configures the EMIF control registers for proper operation: for correct register values see the DSP Initialization section above.

A/D Error Correction

Quixote has been architected without using trimpots for calibration of A/D and D/A scale factors and offsets. Instead, the onboard logic mathematically corrects the results of each analog input channel according to the formula $y = mx + b$ where m is the scale factor (gain) to apply to the input channel and

b is the offset to apply to the input channel. Separate gain and offset values are supported for each analog input channel.

During factory calibration, the optimal coefficients for the gain and offset for each channel have been measured and stored into the flash ROM onboard the Quixote card. These coefficients must be retrieved at runtime and stored into the Quixote gain and offset memory region in the FPGA in order to obtain measurements within the factory-specified accuracy of the analog section. A valid coefficient **must** be provided for each active A/D channel as part of the initialization routine. Calibration coefficients are written into FPGA memory in locations 0x0 through 0xF for channels 0 through 15 respectively for the gain and offset memories.

Table 24. A/D Calibration memory locations and format

Calibration Memory	Address	Channel 1	Channel 0
A/D Gain	0x803E0000	bits 31..16	bits 15..0
A/D Offset	0x80400000	bits 30..16	bits 14..0

The nominal gain of 1 is 0x8000, and 0 is 0 offset. All coefficients are two's complement numbers with a 16 bit gain coefficient and a 14 bit offset.

The Pismo toolset has functions available for use in application software to programmatically adjust the gain and offset of all channels of the analog channels.

A/D Calibration

This section is under construction.

A/D Data Flow Controls and Data Buffering

The A/D sample FIFO provides a 32K sample FIFO directly from the A/D and a 512x64 FIFO after the error correction for interfacing to the 'C6416 EMIF A data bus. Data is deposited in one port of the external FIFO, processed through the error correction, and deposited in the output FIFO by the data acquisition engine in the logic as data is read out by the DSP. The output FIFO has several controls to signal the DSP for data availability, prevent corruption of the data sets, and flush unneeded data. An overflow flag may be monitored that indicates when data has been lost (both the external and internal FIFOs are full and a conversion occurred).

A/D FIFO level status may be monitored by the DSP via CPU reads or may trigger interrupts to the DSP. The A/D FIFO level register may be read by the CPU to determine the exact amount of data currently available in the FIFO. The following diagram gives the register definition.

Bit Number:	31	30	29	28	27	26	25..16	15-0
Bit Field:	AD trigger active	FIFO Overflow	FIFO Too Full	Threshold Level	FULL	EMPT Y	Not used	LEVEL

Figure 43. A/D FIFO Level and Status Register (0x805A0000)

Note: The FIFO has a one-point output register that is not reflected in the LEVEL reported. The FIFO and output buffer are empty when EMPTY is true, add one to the level when EMPTY is false.

(AD trigger active shows, when ‘1’, whether the acquisition trigger is active. See the triggering discussion for more details.)

An interrupt may be signaled to the DSP based on a programmable FIFO level, referred to as the interrupt threshold level. For data acquisition applications, larger packets are normally used to reduce the DSP interrupt rate at the expense of data latency. The programmed threshold is also used as the burst count between interrupt signals. DMA or CPU transfers MUST consume the same number of points as the threshold value before another interrupt will be signaled. This prevents spurious interrupts as the FIFO crosses the threshold value during reads. Burst length is limited to 128 words in the threshold mechanism.

Note: the level must be programmed as number of data points minus one.

Bit Number:	31-7	6-0
Bit Field:	Reserved	Threshold in DWORDS (Number of channel pair samples)

Figure 44. A/D FIFO Interrupt Threshold Level Register (0x80020000)

The primary method used for moving data to the DSP memory is by using a DMA channel. In most cases, DMA delivers data in the most efficient method because it preserves DSP CPU bandwidth, is more efficient at bus utilization, and has the lowest interrupt latency.

It it highly recommended that applications use the DMA-driven functions provided with the Pismo Tool set as the data movement mechanism from the A/D FIFO. Within these drivers, DMA is used on an interrupt-driven basis based upon the data level in the FIFO. These drivers configure the burst length, setup the DMA channel, configure the interrupt for DMA as initialization, then monitor and control the data flow when running.

Single point data reads from the A/D FIFO, using CPU accesses or DMA, may also be performed. Since the A/D FIFO is a burst memory device, the maximum read rate will about 1/3 the speed of larger burst packets due to the transfer setup cycles inherent on the DSP burst memory access protocol.

Monitoring A/D Data Flow Errors

Another important data flow control is the point at which the FIFO is too full to accept another data set. This level is normally programmed as the FIFO depth (512) minus two channels enabled. So, for example, the “too full” FIFO level would be programmed to 248 for an eight channel pair acquisition. Once the “too full” level is reached, no more data points will be stored into the FIFO. This prevents channel pairs from getting dropped, resulting in a corruption of the data ordering that appears to the user that the channels have swapped locations. This value is set in the A/D control register, bits 7..0, and initializes to 248 by default.

If the FIFO ever reaches the condition that it is too full to take another data set, yet another conversion has been triggered, the overflow flag is set true. Since the “too full” mechanism prevents partial data data sets from being written to the FIFO resulting in channel swaps, the entire data set for that conversion will be lost in this case. The overflow flag tells the application that a data set was lost. This will appear as a gap in the data. A record of the time, and data point immediately before the loss, is recorded in the alert log if enabled to do so. The overflow flag is bit 12 of the A/D FIFO status register (0x803D0000).

Once the overflow flag is asserted, it will remain true until it is reset indicating that sometime during the data acquisition process this error condition occurred. The overflow reset is bit 10 in the A/D control register (0x80200000). Setting this bit to ‘1’ resets the flag. It must be set to ‘0’ to enable the mechanism.

Analog Input Timebases and Triggering

Quixote has a variety of triggering modes, sources and controls that enable the programmer to capture A/D samples at the right time under specific conditions. The trigger mechanism has primary concepts of a start trigger, a stop trigger and a

timebase. The start and stop trigger define a region of time when the trigger is active and data may be collected. The timebase defines when the data is sampled during the active trigger region. The following figure shows this concept in detail.

The triggering example shows that once a start trigger falling edge is seen, samples are then collected on rising edges of the timebase. The timebase and triggers need have no special relationship to one another, nor does the timebase need to be periodic. Should a start and stop trigger occur simultaneously, the start trigger is always taken. The sample sets, which are composed of the enabled channel pairs, are collected as is shown during the active trigger interval.

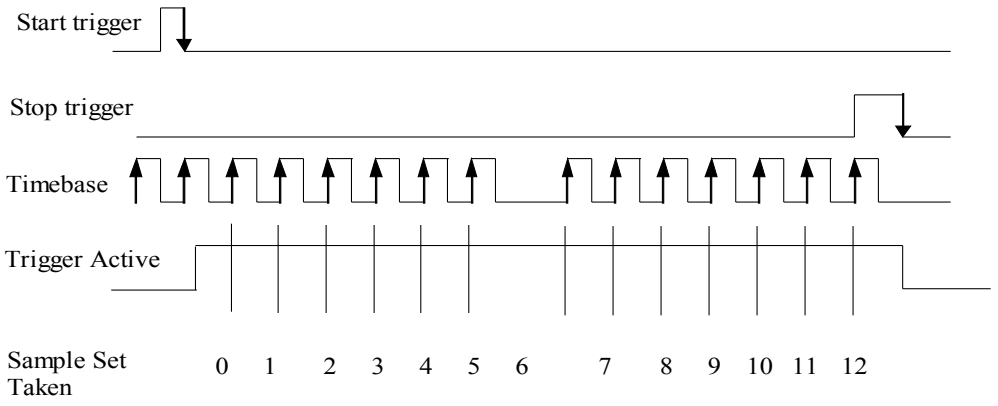


Figure 45. A/D triggering fundamentals

Furthermore, the triggers can be considered to be a single event, referred to a “single-shot”, or can repeatedly fire whenever the setup condition is met.

Software Support

Timebase objects provide a means to collectively configure a clock source, a start trigger and a stop trigger to control the baseboard logic which is used to pace and store the conversions of baseboard analog or digital peripherals. Timebases may thought of as external, independent physical devices like a precision oscillator timebase with programmable start/stop enables. In reality, they control one or more physical resources located on the Velocia or Matador DSP baseboard. However, this portrayal of the timebase as a “virtual” clock source has advantages: For example, the Quixote baseboard contains six programmable timebases, each with different resolutions and capabilities. Which timer should be used for driving a sigma-delta converter? How are they configured when externally gating? The timebase components conceal the complexities of timebase programming by providing a separate component for each clocking technique or mode, so that you may remain blissfully-ignorant of low-level timebase initialization, routing and control mechanics.

It is important to realize that timebase objects merely simplify the programming of baseboard resources. Each timebase is an abstraction of a typical mode of data streaming collected together into an easy-to- operate object. Some typical modes of operation, and their descriptions are tabulated below:

Table 25. Timebase Operation Modes

Operational Mode	Description
Basic	Continuously acquire or generate streams. Acquisition or signal playback commences (is triggered) via the start trigger (by default, software command) and terminates via the stop trigger (default, software command).
Framed	Stream a specified number of samples. Streaming commences via the start trigger (by default, a software command) by and terminates upon having streamed a specified number of samples.
Timed	Stream for a specified time interval. Streaming commences via the start trigger (by default, a software command) and terminates upon having streamed for a specified number of microseconds.
Threshold	Stream whenever a specified threshold conditioned is detected. Streaming commences upon detection of a voltage higher than the threshold value on a specified channel, and terminates whenever the voltage dips below the threshold value.
Slave	Streaming is initiated upon receipt of a start trigger on the SyncLink connector. Streaming is terminated upon receipt of a stop trigger on the Synclink connector. One of the three SyncLink clock pins acts as the conversion clock.
Custom	Streaming is initiated via software command, or detection of a threshold condition. Streaming is terminated via software command, reaching a specified frame count, elapsed time limit or detection of a specified threshold condition. The type, polarity and sense of the start and stop triggers, conversion clock source, frame count, threshold criteria and elapsed time limits are all fully user programmable.

All of the timebases above are used in systems consisting of a one or more DSP baseboards. In multi- target systems, one target application acts as the source for trigger and clock information for all boards in the system. In a system consisting of just a single DSP baseboard, target application programs use any timebase other than a Slave timebase to provide clocking information to its onboard peripheral devices, but this timebases trigger and clock will not be routed externally.

Slave timebases are used exclusively within systems consisting of more than one DSP baseboard. Target applications which are to be slaved to a master within a system each employ a SlaveTmb timebase object which configures its streaming peripherals to receive clock and trigger information from one master baseboard within the system via the SyncLink connector.

Timebase Software Objects

All timebase components derive from the base class BasicTmb. This class provides fundamental services, common to all timebase objects. These services include the ability to:

- Query and change the sample rate at which the timebase will generate conversion pulses
- Initiate a start and stop trigger via software
- Automatically trip the start and stop trigger as the Pismo device driver initiates or terminates streaming

- Derived timebase objects are tabulated below, listed with their default start and stop trigger and conversion clock sources.

Table 26. Analog In Timebase Object Type

Timebase Object	Default Start Trigger	Default Stop Trigger	Default Conversion Clock
BasicTmb	Software	Software	DDS
AdcFramedTmb	Software	A/D Frame Count	DDS
AdcElapsedTmb	Software	A/D Frame Timer	DDS
AdcThreshTmb	Voltage on specific A/D channelgreater than threshold	Voltage on specific A/D channel less than threshold	DDS
MultiTmb	User-specified	User-specified	User-specified
SlaveTmb	SyncLink 2 or 4	SyncLink 1 or 3	SyncLink clock channel

To use a timebase, instantiate the timebase which most closely matches your streaming requirements. Then optionally customize to suit. Finally, attach the timebase to the device driver and begin streaming. For example, the code fragment below illustrates configuration for continuous acquisition from analog input.

```
// Instantiate A/D driver object
AdcStream Ain;

// Attach timebase to A/D driver
BasicTmb Timebase;
Ain.Device().Attach(Timebase);

// Enable specified analog pairs
for (int i = 0; i < NumChannels; ++i)
    Ain.Device().Channels().Enabled(i, Status.ChannelMask&(1<<i));

// Size the stream buffers to hold specified number of events
Ain.Events(NumEvents);

// Open the analog stream objects
Ain.Open();
```

To customize the above code to use an external clock source, instead of the default baseboard DDS, simply manipulate the Timebase object dynamically after creation:

```
// Attach timebase to A/D driver
ContinuousTmb Timebase;
Timebase.ClockSource(csExternal);
Ain.Device().Attach(Timebase);

...

// Open the analog stream objects
Ain.Open();
```

The MultiTmb object provides a generic interface to all of the timebase circuitry on the baseboard. All of the features and properties of the baseboard timers are exposed, providing a complex, albeit unrestricted view of the capabilities of the baseboard timebase facilities. In general, it is easier to use one of the focused timebase components, such as ContinuousTmb or FramedTmb as the trigger source within application programs. However, in some rare instances, an unusual combination

of features of the onboard timebase circuitry is not exposed or accessible using the simplified timebase components. MultiTmb is provided to address those esoteric needs. While this timebase is the most flexible, it is also the most difficult to configure.

Hardware Support

A/D Clock Sources

The clock source may be chosen from several sources, both on-card and external, that allow precision sampling based upon the flexible direct digital synthesis (DDS), DSP timers, external input, or PXI triggers/clocks for multiple card synchronization. The timebase need not be periodic, but AC performance will be degraded for non-periodic timebases. Both A/D channels run using the same timebase and are synchronous conversions.

For time driven acquisitions, the DDS offers the most flexible timebase option. This is a programmable timebase, derived from a stable 20 MHz TCXO clock source, that can generate any frequency 0-105 MHz, with 0.02 Hz resolution. The DSP timers offer two more timebases, that may be used for time- driven acquisitions, including sub-divided output of the DDS or external signal.

External inputs to the external clock should be TTL levels (‘1’ >2.4V, ‘0’ < 0.7V) and may be 50 ohm terminated or not to match the signal source impedance. Multicard systems may share the timebase using the SyncLink or ClockLink timebase source. See the discussion on multicard synchronization in this manual.

Table 27. A/D conversion timebase (0x64130000)

Bit	Timebase selected
0	not used
1	DDS, ‘0’ = not selected, default
2	DSP timer 0, ‘0’ = not selected, default
3	DSP timer 1, ‘0’ = not selected, default
4	External Clock, ‘0’ = not selected, default
5	PXI 0, ‘0’ = not selected, default
6	PXI 1, ‘0’ = not selected, default
7	PXI 2, ‘0’ = not selected, default

Start and Stop Trigger Selections

The start and stop triggers may be chosen from a number of sources, both on-card and external, that allow a great amount of flexibility in controlling the data sampling period. Furthermore, the trigger control logic may be programmed as either edge or level sensitive, with programmable polarity control.

- The types of triggers supported include the following:
    - Software triggers - Issue a start or stop from software. These are available in addition to the other trigger source selection, so that software may always start or stop data collection.
    - External Input - Use any TTL signal as a start or stop trigger
    - SyncLink Inputs - use SyncLink for Multi-card synchronization by sharing trigger sources

- Frame Timer - Collect data for a specified period of time, programmable in 1 uS intervals.
- Frame Counter - Collect data for a specified number of samples. Useful for algorithms like FFTs, where a certain number of points is required for a data set.
- Analog Threshold - Use any A/D channel as selected by programmable control, to control the data collection interval.
- Always, Never - Useful when you never want to stop, or always begin without qualification.

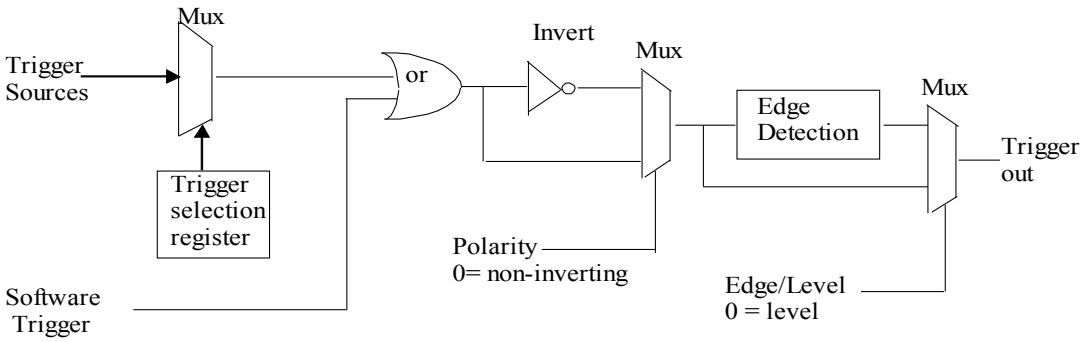
Table 28. A/D start trigger selection register (0x80100000)

Bit	Function	Purpose
0	External ADC Start Trigger	External signal input
1	not used	-
2	PXI 2	Multicard Synchronization
3	PXI 3	Multicard Synchronization
4..7	Not used	-
8	Analog Threshold on threshold channel	Trigger on an A/D channel input signal at a programmable threshold
9..13	not used	
14	Always	Always start, without qualification

Table 29. A/D start trigger selection register (0x80100000)

Bit	Function	Purpose
0	Not Used	-
1	External ADC Stop Trigger	External signal input
2	PXI 2	Multicard Synchronization
3	PXI 3	Multicard Synchronization
4..7	not used	-
8	Analog Threshold on threshold channel	Trigger on an A/D channel input signal at a programmable threshold
9	Frame Timer	Stop collection after a specified time interval
10	Frame Counter	Stop collection after a specified number of points
11..14	Not Used	-
15	Never	Never stop, without qualification

The trigger selection and conditioning consists of logic allowing the selection of a trigger source, polarity selection, and edge or level control. The internal logic triggers on rising edges of the trigger out signal, so the trigger signals should be conditioned to so once through the logic.



Bit	Function
0	Trigger polarity 0=non-inverting, default
1	Level or edge sensitivity; 0 = level, default

Figure 46. A/D Trigger polarity and type selection register (start = 0x80120000, stop = 0x80160000)

Threshold Triggering

Monitoring the level of an incoming analog signal, and collecting data based upon that level, is performed using the analog threshold triggering mechanism. The threshold triggering mechanism allows one channel to be observed, as selected by the threshold channel register, that is used as a start trigger, stop trigger or both. The threshold level is also programmable. Since the threshold trigger further enters the trigger conditioning logic, it is also possible to trigger when the signal is under a threshold to stop acquisition.

To prevent false triggering from noise, there is a programmable hysteresis for the trigger mechanism. The hysteresis value determines the number of points, at or above the qualifying threshold, the must be encountered consecutively to consider this a valid threshold trigger. Likewise, once a threshold trigger has been asserted, it cannot deassert until it is below the threshold for the hysteresis number of points. The hysteresis value can be up to 8 points, while a number of 0 points results in triggering each time the threshold is exceeded. Depending on the nature of the signal used for threshold triggering, the hysteresis value should be according to the expected noise level, sampling rate, and maximum rate of change expected in the signal to optimize noise rejection yet capture fast transient peaks.

The threshold is set as a 14-bit, two's complement value in threshold trigger register. The logic performs a comparison of the digitized A/D value on the selected threshold channel to the threshold value. If the analog value is at or above the threshold value it is counted for the hysteresis mechanism.

Note: The threshold trigger will not trigger until after the first 16 analog samples on the selected channel have passed.

Table 30. Threshold Channel and Hysteresis Register (0x801A0000)

Bit	Function
31..19	Not Used
18..16	Hysteresis value

Bit	Function
15..2	Not used
1	Monitor Channel 1
0	Monitor Channel 0

Table 31. Threshold value register (0x80180000)

Bits	Function
31..14	not used
13..0	Threshold Value

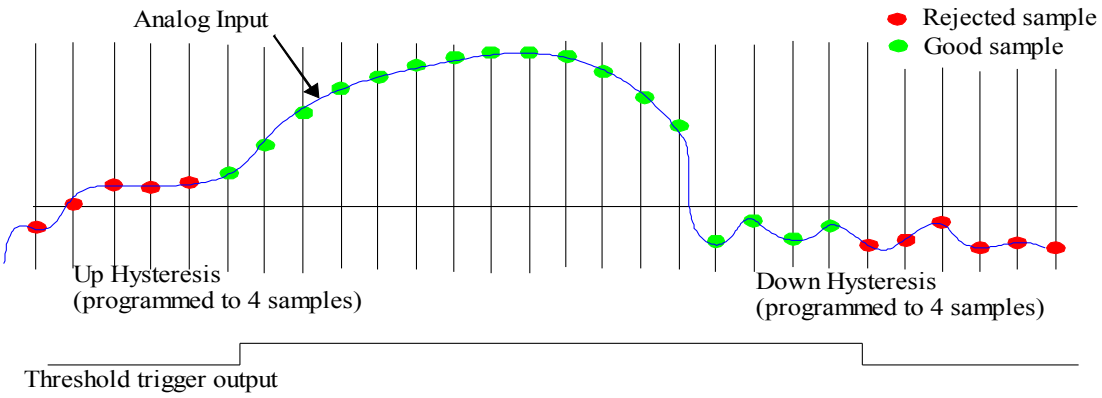


Figure 47. Threshold trigger example with programmable hysteresis of 4 samples

A/D Frame Timer

Quixote can also acquire data for a precision time period after a start trigger is asserted. The data that is acquired in this mode, referred to as a frame of data, will be acquired until the programmable frame timer expires. The time period is programmed in 0.1 uS intervals, with up to 2^24 us (approximately 1.677 us) in any frame with an accuracy of -0, +0.1 us. Longer timed intervals generally do not need precision to 0.1 us, and at therefore best handled with the DSP timers.

The frame timer is only available as a stop trigger.

Table 32. A/D Frame Timer Register (0x801C0000)

Bits	Function
31..24	not used
23..0	Frame timer value

A/D Frame Counter

Another method used for data collection and analysis is to collect an exact number of points. This may be particularly useful in algorithms that require a specific data set size, such as FFTs, or in measurements dependent on aperiodic timebases such as measuring every 1 degree of a shaft rotation. The A/D frame counter allows the logic to produce stop trigger at an exact number of points, all under programmable control.

The frame counter mechanism is also used in the alert mechanism to report the number of points collected during any acquisition process. Since the frame counter simply counts the number of timebase clocks that occurred in an active trigger, it increments once for each data set acquired.

The frame counter is a 32-bit counter, with a rollover alert that may be enabled in the alert log. When the counter is not being used to produce a stop trigger, it is advised that the maximum value be loaded in the frame count, so that rollover alerts are triggered as infrequently as possible.

The frame counter is only available as a stop trigger. Turning off the acquisition engine, A/D RUN = '0', results in the frame timer being reset to 1.

Table 33. A/D Frame Counter Register (0x801E0000)

Bits	Function
31..0	Frame timer value

Single-Shot Mode
Single-shot mode allows for capture of single event beginning when the start trigger, as selected and conditioned by the trigger logic, is seen until a stop trigger is fired. Subsequent start triggers will be ignored, even if they meet the conditions as prescribed in the logic, until the single-shot mechanism is reset by the DSP. This allows the application to capture a single trigger event, as is common on digital oscilloscopes.

Single-shot mode is enabled in the A/D control register (0x80000000), bit 14. Setting the single-shot bit to '1' enables the single-shot mode. After the single-shot has fired, it must be re-armed by setting this bit to '1' again.

Pretrigger Mode
Pretrigger mode allows the DSP to capture data before the start trigger condition is satisfied. The pretrigger allows the data to be collected and flow to the DSP, just as though the trigger period is active, so that the DSP application can save the amount of data before the start trigger needed for pretrigger data. The pretrigger period is completed when the start trigger fires. Software should enable the start trigger alert so that it can manage the data buffers for the pretrigger data. Within the alert message, the frame count indicates the number of the data point where the start trigger occurred, thus allowing the application to know where the start trigger fired, and the pretrigger ended.

Pretrigger mode is enabled in the A/D control register (0x80000000), bit 16. Setting the pretrigger bit to '1' enables the pretrigger mode. After the start trigger has fired, the pretrigger mode is considered complete and it must be re-armed by setting this bit to '1' again.

Analog Input Comparator Trigger

Quixote has a programmable threshold analog comparator input that may be used as a trigger for analog capture and playback. The analog comparator trigger is useful for detecting the level of a signal, such as the output of a signal power monitor, for gain control and monitoring. Within the triggering mechanisms, the analog comparator trigger may be configured as a level or edge trigger, with programmable polarity. Either of the triggering mechanisms for DACs or A/Ds may use this trigger as a start or stop trigger.

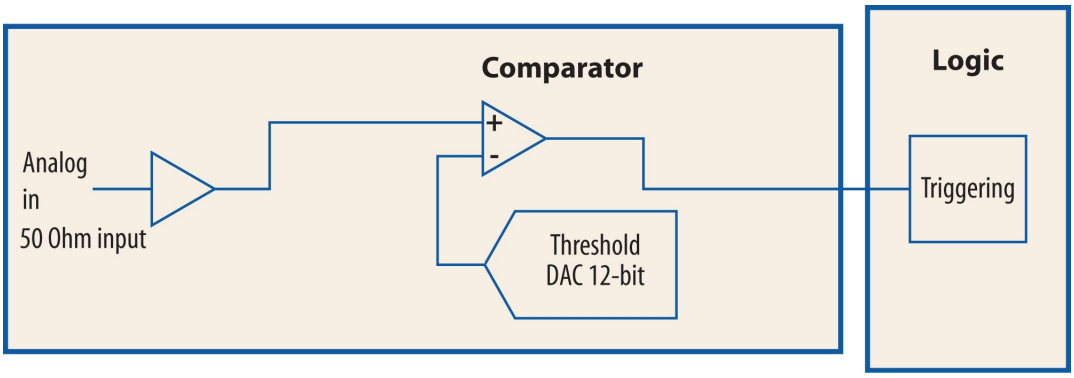


Figure 48. Input Comparator Trigger

Inputs above the threshold voltage as set by the DAC result in a positive logic '1'; outputs below are '0'. The comparator has an approximate response time of 4.5 ns. The input range of the analog comparator is +/-2VDC peak-to-peak. This input is on connector J2 and is a 50 ohm input.

Software functions in the Pismo Toolset are provided to adjust the threshold level and configure the triggering matrices.

Analog Outputs

Quixote implements two channels of analog output that allow simultaneous outputs at 105 MHz with 14-bit resolution. Each channel is an independent signal chain, with no signal multiplexing, allowing for true simultaneous output updates and better channel isolation. These DACs are optimized for transmit channel applications and are excellent for high speed waveform generation. Both channels must run synchronously in the standard logic, although a wide variety of trigger mechanisms and timebases have been provided.

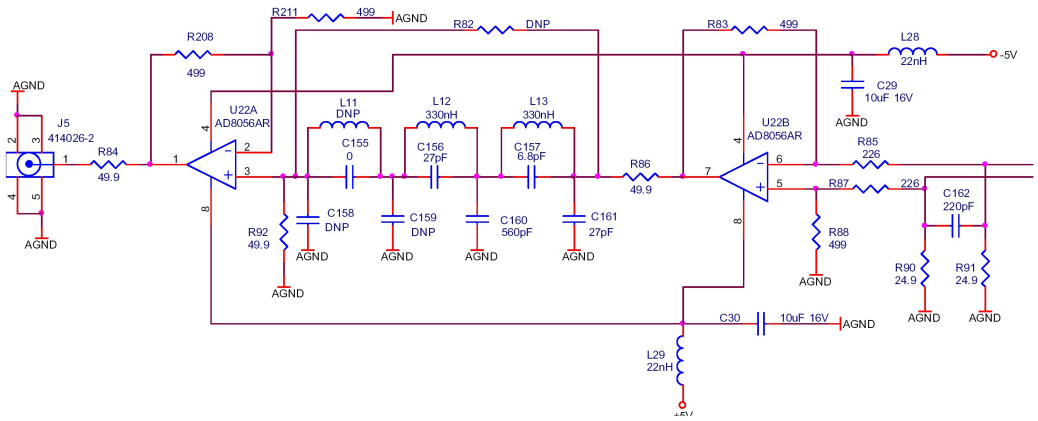


Figure 49. DAC Output Filtering

All channels have an output range of +/-1 V. A multipole anti-alias filter follows each D/A converter that has its -3dB point set to 10.1 MHz. Custom ranges and filters may be special ordered, contact Innovative Integration sales department.

Data collection is controlled by the triggering mechanisms. Triggering is defined as the time period for data collection bounded by a start trigger and stop trigger. During the active trigger period, the timebase defines when the samples are acquired. Various sources for start trigger and stop trigger including external inputs allow a variety of data collection methods to be used.

The analog outputs are digitally corrected for gain and offset errors in the FPGA using a first order model. The error correction is done real-time in the FPGA as each data point is moved to the external sample FIFO before the D/As. Calibration coefficients are determined either at factory calibration or derived from the auto-calibration of the card are used in the error correction. The coefficients are stored in an on-card non-volatile memory and must be loaded into the error correction logic before use. This is further discussed in the calibration section.

The D/A points are held in a 512 sample FIFO directly from the DSP EMIF A bus, as data buffering for the DSP. These points are written by the DSP as four, 14-bit stacked in the 64-bit data word. Each 64-bit number has two pairs of two's-complement numbers stacked with channel 0 oldest sample on bytes 1-0, channel 1 on bytes 3-2, and the newer sample pair stacked on bytes 5-4 for channel 0 and 7-6 for channel 1. These points are written by the DSP as DMA or CPU reads, using programmable level interrupts from the FIFO to pace the data flow. The logic then processes each data point to correct errors and deposits it into the appropriate large external sample FIFO (32K samples) for each channel.

The following block diagram shows the functional interconnections of the analog output subsystem.

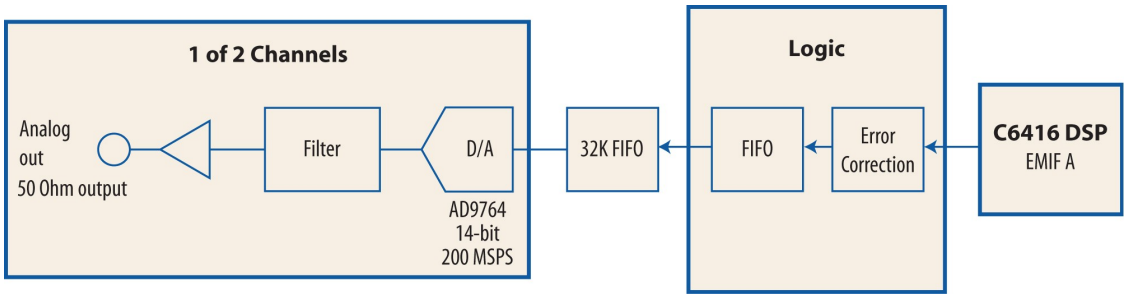


Figure 50. D/A Output Block Diagram

Smoothing Filter Characteristics

The analog output smoothing filter provides signal reconstruction and helps to eliminate out of band noise. The corner frequency, defined as the -3 dB point, is set to 10.1 MHz. The following figures show the filter response from 100 kHz to 100 MHz, and a detail of the region from 1 MHz to 50 MHz.

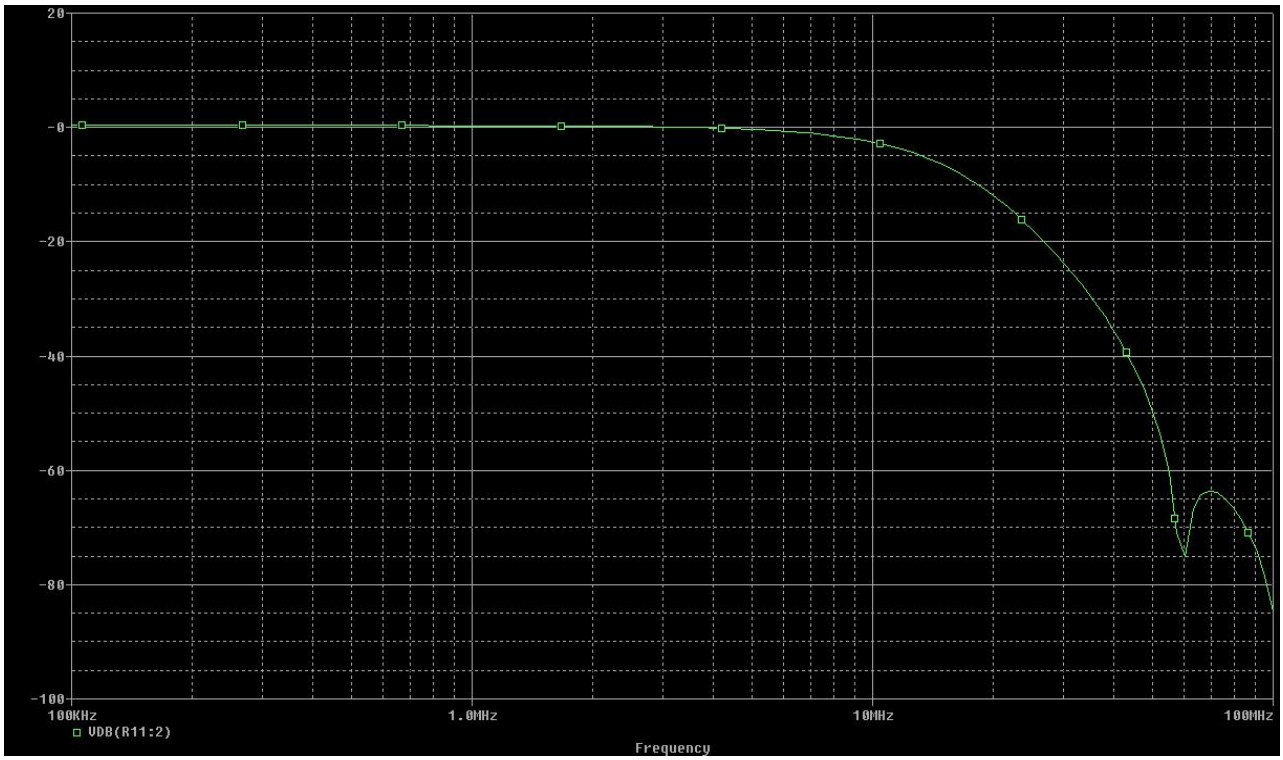


Figure 51. DAC smoothing filter frequency response

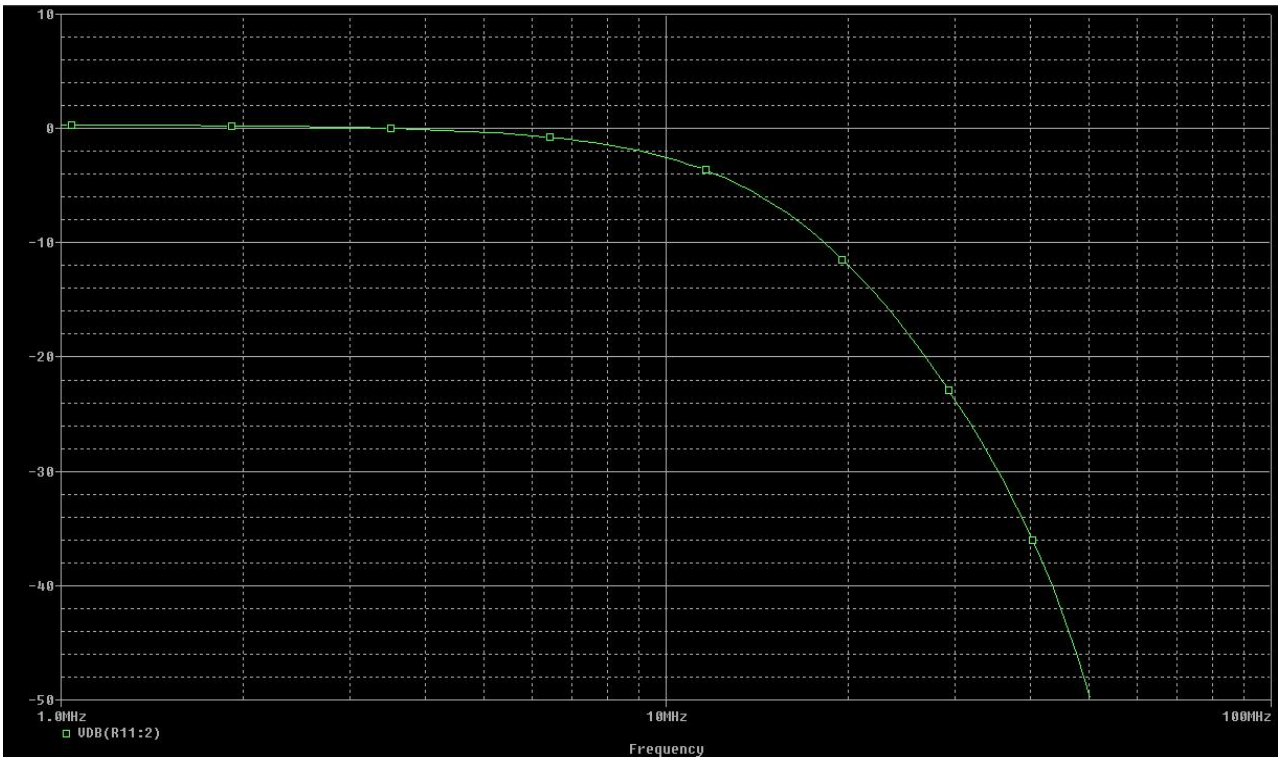


Figure 52. Detail of the DAC smoothing filter frequency response

D/A Triggering and Data Collection

The D/A data conversion engine in the FPGA has the primary task of moving data to the DACs based on the triggering conditions set by the DSP. This allows the programmer to update the DACs at very precise times, under specified conditions, that help the programmer to control the conversion process timing and data rates, as required by the application. The DSP controls the data conversion process by enabling the data conversion, initializing triggering parameters, and moving the data to the FIFO as demanded by the conversion process.

Enabling The D/A data conversion engine

The data acquisition engine is enabled by the DAC RUN bit in the analog control register (0x80000000, bit 1). The DAC RUN bit, when set to '1', enables the data conversion engine to move data when the triggering conditions are satisfied. The DAC RUN bit must be true for triggers to be recognized, for the data to flow, and essentially acts as a reset to the D/A data conversion process and triggering mechanism. Triggers may be cleared and the data engine reset by making DAC RUN bit false. The fifo must be cleared separately by the DAC FIFO RESET bit.

D/A Data Format

Data must always be written to the DAC input FIFO as sets in ascending channel order, with the newest data pair on the upper half of the 64-bit output word. Figure 19 shows the data word format and the data set accumulating over time in the FIFO. All data is two's-complement numbers 16-bit numbers, with 14 bit precision. Samples to be played at time t comprise the first data set, with the channels ordered as shown, samples to be played at time t+1 comprise the second data set. This format continues *ad infinitum*.

Table 34. D/A channel ordering and data set format

Sample time	Byte 7-6	Byte 5-4	Byte 3-2	Byte 1-0
t+1, t	channel 1 @ t+1	channel 0 @ t+1	channel 1 @ t	channel 0 @ t
t+3, t+2	channel 1 @ t+3	channel 0 @ t+3	channel 1 @ t+2	channel 0 @ t+2
t+(n+1), t+n	channel 1 @ t+(n+1)	channel 0 @ t+(n+1)	channel 1 @ t+n	channel 0 @ t+n

Since the DSP interacts primarily with the FIFO, which has a very high speed burst memory interface capable of 1GB/sec bursts, Quixote is able to support high data rates with minimal bus loading - leaving more CPU time for computation. In order to optimize data transfer rates to the D/A FIFO, D/A data is written to the FIFO using synchronous access cycles on the 'C6416 EMIF A interface. This allows one word, comprised of 2 data samples per channel, to be written to the FIFO per EMIF clock cycle in burst transactions after a 2 clock setup per burst transfer. The DSP external memory interface must be initialized as a burst RAM (SBSRAM) memory space for proper operation. Quixote's Pismo Toolset software automatically configures the EMIF control registers for proper operation: for correct register values see the DSP Initialization section above.

D/A Error Correction

Quixote has been designed without hardware trim pots for calibration of A/D and D/A scale factors and offsets. Instead, the on board logic mathematically corrects the results of each analog input channel according to the formula $y = mx + b$ where m is the scale factor (gain) to apply to the input channel and

b is the offset to apply to the input channel. Separate gain and offset values are supported for each analog input channel.

During factory calibration, the optimal coefficients for the gain and offset for each channel have been measured and stored into the flash ROM on board the Quixote card. These coefficients must be retrieved at runtime and stored into the Quixote gain and offset memory region in the FPGA in order to obtain measurements within the factory-specified accuracy of the analog section. A valid coefficient must be provided for each active D/A channel as part of the initialization routine. Calibration coefficients are written into FPGA register with the format shown in figure 37.

Table 35. D/A calibration memory registers and format

Calibration Memory	Address	Channel 1	Channel 0
D/A Gain	0x80420000	bits 31..16	bits 15..0
D/A Offset	0x80440000	bits 30..16	bits 14..0

The nominal gain of 1 is 0x8000, and 0 is 0 offset. All coefficients are two's complement numbers with a 16-bit gain coefficient and a 14-bit offset.

D/A Calibration

This section is under construction.

D/A Data Flow Controls and Data Buffering

The D/A data streams use two FIFOs to buffer data : an input FIFO provides a 512x64 buffer directly from the DSP bus, and an output sample FIFO with 32K sample capacity. Data is deposited in the input FIFO by the DSP using EMIF A as space permits and according to pacing flags, and is processed by the D/A conversion engine as space in the output sample FIFO permits. The input FIFO has several controls signal the DSP for that data is required, prevent corruption of the data sets, and flush unneeded data.

D/A FIFO level status may be monitored by the DSP via CPU reads or may trigger interrupts to the DSP. The D/A FIFO level register may be read by the CPU to determine the exact amount of data currently available in the FIFO. The following diagram gives the register definition.

Bit Number:	31	30	29	28	27	26	25..16	15-0
Bit Field:	D/A trigger active	FIFO Under flow	FIFO starving	Threshold Level	FULL	EMPT Y	Not used	LEVEL

Figure 53. D/A FIFO Level and Status Register (0x805C0000)

Note: The FIFO level is the sum of the input and output FIFOs. Points in the error correction pipe are not in this sum; there may be as many as 7 points in-transit to the output sample FIFO at any time.

Table 36. D/A FIFO Level and Status Register Field Description

DAC Status	Function
LEVEL	How many points are in the DAC FIFOs. This is the sum the points in the input FIFO (0-511) and the output sample FIFO (0-32767).
Empty	The input FIFO is empty when ‘1’
Full	The input FIFO is full when ‘1’
Threshold Level	The FIFO LEVEL is above programmed threshold when ‘1’
FIFO Starving	The input FIFO does not have enough points to run the pipe when ‘1’

DAC Status	Function
FIFO Underflow	The DAC needed a data point and none were available in the output sample FIFO - this indicates an error when ‘1’.
D/A Trigger Active	The D/A trigger is active when ‘1’

An interrupt may be signalled to the DSP based on a programmable FIFO level, referred to as the interrupt threshold level. For data playback and waveform generation applications, large packets may be used to reduce the DSP interrupt rate at the expense of data latency. The programmed threshold is also used as the burst count between interrupt signals. DMA or CPU transfers *MUST* consume the same number of points as the threshold value before another interrupt will be signalled. This prevents spurious interrupts as the FIFO crosses the threshold value during reads.

Note: the level must be programmed as number of data points minus one.

Bit Number:	31-7	6-0
Bit Field:	Reserved	Threshold in DWORDS (Number of channel pairs enabled)

Figure 54. D/A FIFO Interrupt Threshold Level Register (0x800E0000)

The primary method used for moving data from the DSP memory is by using a DMA channel. In most cases, DMA delivers data in the most efficient method because it preserves DSP CPU bandwidth, is more efficient at bus utilization, and has the lowest interrupt latency.

It it highly recommended that applications use the DMA-driven functions provided with the Pismo Tool set as the data movement mechanism from the D/A FIFO. Within these drivers, DMA is used on an interrupt-driven basis based upon the data level in the FIFO.

Single point data write to the D/A FIFO, using CPU accesses or DMA, may also be performed. Since the D/A FIFO is a burst memory device, the maximum write rate will about 1/3 the speed of larger burst packets due to the transfer setup cycles inherent on the DSP burst memory access protocol.

Monitoring D/A Data Flow Errors

If the FIFO ever reaches the condition that it requires a data sample and none is available, yet another conversion has been triggered, the underflow flag is set true. The logic prevents partial data sets taken from the FIFO resulting in channel swaps, so the entire conversion will be skipped in this case. The underflow flag tells the application that a data conversion was skipped. This will appear as a gap in the output waveform. A record of the time, and data point immediately before the loss, is recorded in the alert log if enabled to do so. The underflow flag is bit 30 of the D/A FIFO status register (0x800E0000).

Once the underflow flag is asserted, it will remain true until it is reset indicating that sometime during the data acquisition process this error condition occurred. The overflow reset is bit 13 in the D/A control register (0x80080000). Setting this bit to ‘1’ resets the flag. It must be set to ‘0’ to enable the underflow mechanism.

Analog Output Timebases and Triggering

Quixote has a variety of triggering modes, sources and controls that enable the programmer to convert D/A samples at the right time under specific conditions. The trigger mechanism has a primary concepts of a start trigger, a stop trigger, and a timebase. The *start* and *stop trigger* define a region of time when the trigger is active and data may be converted. The *timebase* defines when the data is converted during the active trigger region. The following figure shows this concept in detail.

The triggering example shows that once a start trigger falling edge is seen, samples are then collected on rising edges of the timebase. The timebase and triggers need have no special relationship to one another, nor does the timebase need to be periodic. If a start and stop trigger occur simultaneously, then the start trigger always prevails. The sample sets, which are composed of the enabled channel pairs, are collected as is shown during the active trigger interval.

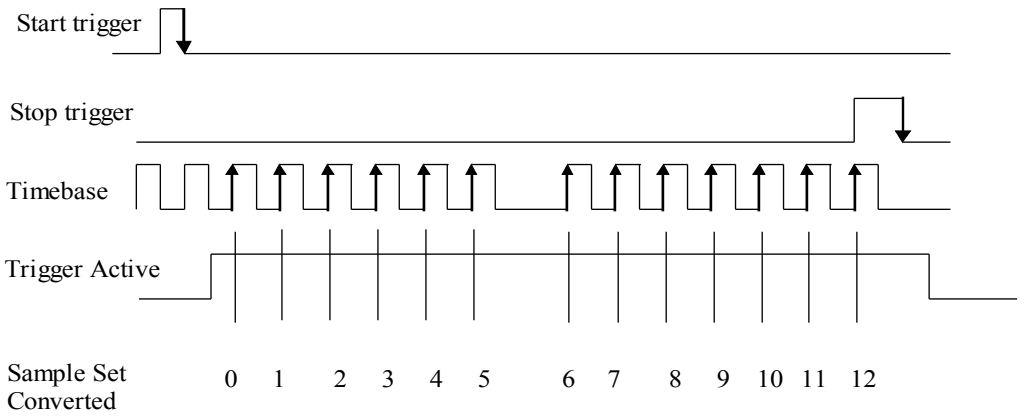


Figure 55. DAC triggering Fundamentals

Software Support

Timebase objects provide a means to collectively configure a clock source, a start trigger and a stop trigger to control the baseboard logic which is used to pace and store the conversions of baseboard analog or digital peripherals. Timebases may thought of as external, independent physical devices like a precision oscillator timebase with programmable start/stop enables. In reality, they control one or more physical resources located on the Velocia and Matador DSP baseboards. However, this portrayal of the timebase as a “virtual” clock source has advantages: For example, the Toro baseboard contains six programmable timebases, each with different resolutions and capabilities. Which timer should be used for driving a sigma-delta converter? How are they configured when externally gating? The timebase components conceal the complexities of timebase programming by providing a separate component for each clocking technique or mode, so that you may remain blissfully-ignorant of low-level timebase initialization, routing and control mechanics.

It is important to realize that timebase objects merely simplify the programming of baseboard resources. Each timebase is an abstraction of a typical mode of data streaming collected together into an easy-to- operate object. Some typical modes of operation, and their descriptions are tabularized below:

Table 37. Timebase Operation Modes

Operational Mode	Description
Basic	Continuously acquire or generate streams. Acquisition or signal playback commences (is triggered) via the start trigger (by default, software command) and terminates via the stop trigger (default, software command).
Framed	Stream a specified number of samples. Streaming commences via the start trigger (by default, a software command) by and terminates upon having streamed a specified number of samples.
Timed	Stream for a specified time interval. Streaming commences via the start trigger (by default, a software command) and terminates upon having streamed for a specified number of microseconds.
Threshold	Stream whenever a specified threshold conditioned is detected. Streaming commences upon detection of a voltage higher than the threshold value on a specified channel, and terminates whenever the voltage dips below the threshold value.
Slave	Streaming is initiated upon receipt of a start trigger on the SyncLink connector. Streaming is terminated upon receipt of a stop trigger on the Synclink connector. One of the three SyncLink clock pins acts as the conversion clock.
Custom	Streaming is initiated via software command, or detection of a threshold condition. Streaming is terminated via software command, reaching a specified frame count, elapsed time limit or detection of a specified threshold condition. The type, polarity and sense of the start and stop triggers, conversion clock source, frame count, threshold criteria and elapsed time limits are all fully user programmable.

All of the timebases above are used in systems consisting of a one or more DSP baseboards. In multi-target systems, one target application acts as the source for trigger and clock information for all boards in the system. In a system consisting of just a single DSP baseboard, target application programs use any timebase other than a Slave timebase to provide clocking information to its onboard peripheral devices, but this timebases trigger and clock will not be routed externally.

Slave timebases are used exclusively within systems consisting of more than one DSP baseboard. Target applications which are to be slaved to a master within a system each employ a SlaveTmb timebase object which configures its streaming peripherals to receive clock and trigger information from one master baseboard within the system via the SyncLink connector.

Timebase Software Objects

All timebase components derive from the base class BasicTmb. This class provides fundamental services, common to all timebase objects. These services include the ability to:

- Query and change the sample rate at which the timebase will generate conversion pulses
- Initiate a start and stop trigger via software
- Automatically trip the start and stop trigger as the Pismo device driver initiates or terminates streaming

Derived timebase objects are tabularized below, listed with their default start and stop trigger and conversion clock sources.

Table 38. Analog Out Timebase Object Types

Timebase Object	Default Start Trigger	Default Stop Trigger	Default Conversion Clock
BasicTmb	Software	Software	DDS
DacFramedTmb	Software	D/A Frame Count	DDS
DacElapsedTmb	Software	D/A Frame Timer	DDS
DacDelayTmb	Software	Software	DDS
MultiTmb	User-specified	User-specified	User-specified
SlaveTmb	SyncLink 2 or 4	SyncLink 1 or 3	SyncLink clock channel

To use a timebase, instantiate the timebase which most closely matches your streaming requirements. Then optionally customize to suit. Finally, attach the timebase to the device driver and begin streaming. For example, the code fragment below illustrates configuration for continuous acquisition from analog input.

```
// Instantiate A/D driver object
AdcStream Ain;

// Attach timebase to A/D driver
BasicTmb Timebase;
Ain.Device().Attach(Timebase);

// Enable specified analog pairs
for (int i = 0; i < NumChannels; ++i)
    Ain.Device().Channels().Enabled(i, Status.ChannelMask&(1<<i));

// Size the stream buffers to hold specified number of events
Ain.Events(NumEvents);

// Open the analog stream objects
Ain.Open();
```

To customize the above code to use an external clock source, instead of the default baseboard DDS, simply manipulate the Timebase object dynamically after creation:

```
// Attach timebase to A/D driver
ContinuousTmb Timebase;
Timebase.ClockSource(csExternal);
Ain.Device().Attach(Timebase);

...

// Open the analog stream objects
Ain.Open();
```

The MultiTmb object provides a generic interface to all of the timebase circuitry on the baseboard. All of the features and properties of the baseboard timers are exposed, providing a complex, albeit unrestricted view of the capabilities of the baseboard timebase facilities. In general, it is easier to use one of the focused timebase components, such as ContinuousTmb or FramedTmb as the trigger source within application programs. However, in some rare instances, an unusual combination of features of the onboard timebase circuitry is not exposed or accessible using the simplified timebase components. MultiTmb is provided to address those esoteric needs. While this timebase is the most flexible, it is also the most difficult to configure.

Hardware Support

D/A Clock Sources

The timebase may be chosen from several sources, both on-card and external, that allows precision conversions based upon the flexible direct digital synthesis (DDS), DSP timers, external input, or PXI triggers/clocks for multiple card synchronization. The timebase need not be periodic. All D/A channels run using the same timebase and are synchronous conversions. The DACs also have a delayed timebase mode for servo applications (described in the next section).

For time driven conversions, the DDS offers the most flexible timebase option. This is a programmable timebase, derived from a stable 20 MHz TCXO clock source, that can generate a timebase at any frequency 0-105 MHz, with 0.02 Hz resolution. The DSP timers offer two more timebases, that may be used for time-driven acquisitions, including sub-divided output of the DDS or external signal.

External inputs to the external clock should be TTL levels (‘1’ >2.4V, ‘0’ < 0.7V) and may be 50 ohm terminated or not to match the signal source impedance. Multicard systems may share the timebase using the PXI inputs. See the discussion on multicard synchronization in this manual.

Table 39. D/A conversion timebase (0x64130000)

Bit	Timebase selected
16	not used
17	DDS, ‘0’ = not selected, default
18	DSP timer 0, ‘0’ = not selected, default
19	DSP timer 1, ‘0’ = not selected, default
20	External Clock, ‘0’ = not selected, default
21	PXI 0, ‘0’ = not selected, default
22	PXI 1, ‘0’ = not selected, default
23	PXI 2, ‘0’ = not selected, default

Start and Stop Triggers

The start and stop triggers may be chosen from a number of sources, both on- card and external, that allow a great amount of flexibility in controlling the data sampling period. Furthermore, the trigger control logic may be programmed as either edge or level sensitive, with programmable polarity control.

The types of triggers supported include the following:

- Software triggers - Issue a start or stop from software. These are available in addition to the other trigger source selection, so that software may always start or stop data collection.
- External Input - Use any TTL signal as a start or stop trigger
- SyncLink Inputs - use SyncLink for Multi-card synchronization by sharing trigger sources
- Frame Timer - Collect data for a specified period of time, programmable in 1 uS intervals.
- Frame Counter - Collect data for a specified number of samples. Useful for algorithms like FFTs, where a certain number of points is required for a data set.

- Always, Never - Useful when you never want to stop, or always begin without qualification.

Table 40. D/A start trigger selection register (0x80200000)

Bit	Function	Purpose
0	External DAC Start Trigger	External signal input
1	not used	-
2	PXI 2	Multicard Synchronization
3	PXI 3	Multicard Synchronization
4..13	not used	-
14	Always	Always start, without qualification

Table 41. D/A stop trigger selection register (0x80240000)

Bit	Function	Purpose
0	Not Used	-
1	External DAC Stop Trigger	External signal input
2	PXI 2	Multicard Synchronization
3	PXI 3	Multicard Synchronization
4..8	not used	-
9	Frame Timer	Stop collection after a specified time interval
10	Frame Counter	Stop collection after a specified num ber of points
11..14	Not Used	-
15	Never	Never stop, without qualification

The trigger selection and conditioning consists of logic allowing the selection of a trigger source, polarity selection, and edge or level control. The internal logic triggers on falling edges of the trigger out signal, so the trigger signals should be conditioned to so once through the logic.

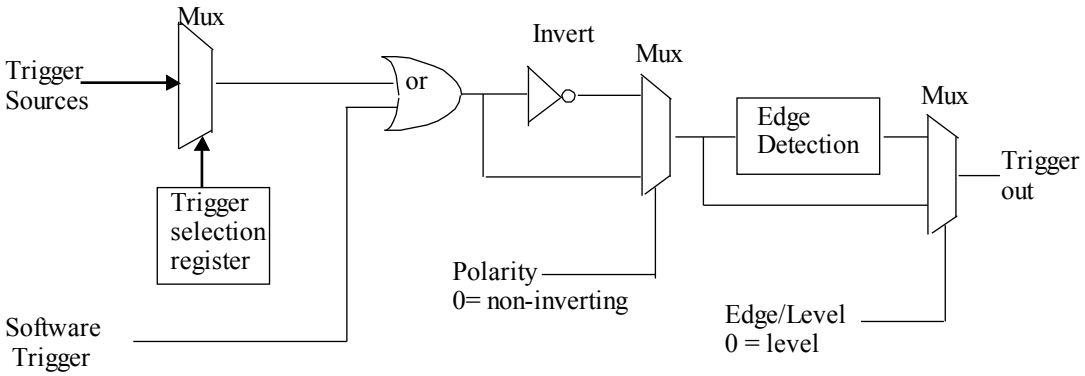


Figure 56. D/A Trigger Selection Logic

Table 42. D/A Trigger polarity and type selection register (start = 0x80220000 stop = 0x80260000)

Bit	Function
0	Trigger polarity 0=non-inverting, default
1	Level or edge sensitivity; 0 = level, default

D/A Frame Timer

Quixote can also perform D/A conversions for a precision time period after a start trigger is asserted. The data that is converted in this mode, referred to as a frame of data, will be converted until the programmable frame timer expires. The time period is programmed in 0.1 uS intervals, with up to 2^24 us (approximately 1.677 seconds) in any frame with an accuracy of -0, +0.1 us. Longer timed intervals generally do not need precision to 0.1 us, and at therefore best handled with the DSP timers.

The frame timer is only available as a stop trigger.

Table 43. D/A Frame Timer Register (0x802C0000)

Bits	Function
31..24	not used
23..0	Frame timer value

D/A Frame Counter

Another method used for generating precision outputs is to convert an exact number of points after a start trigger. This may be particularly useful in waveform generation where an exact number of points are used in the waveform model. The D/A frame counter allows the logic to produce stop trigger at an exact number of points, all under programmable control.

The frame counter mechanism is also used in the alert mechanism to report the number of points consumed during any conversion process. Since the frame counter simply counts the number of timebase clocks that occurred in an active trigger, it increments once for each data set acquired.

The frame counter is a 32-bit counter, with a rollover alert that may be enabled in the alert log. When the counter is not being used to produce a stop trigger, it is advised that the maximum value be loaded in the frame count, so that rollover alerts are triggered as infrequently as possible.

The frame counter is only available as a stop trigger. Turning off the acquisition engine, D/A RUN = ‘0’, results in the frame timer being reset to 1.

Table 44. D/A Frame Counter Register (0x802E0000)

Bits	Function
31..0	Frame timer value

Single-Shot Mode

Single-shot mode allows for generation of a single frame of conversions beginning when the start trigger, as selected and conditioned by the trigger logic, is seen until a stop trigger is fired. Subsequent start triggers will be ignored, even if they meet the conditions as prescribed in the logic, until the single-shot mechanism is reset by the DSP. This allows the application to generate a single set of data for the event, for control of the D/A conversion process.

Single-shot mode is enabled in the D/A control register (0x80080000), bit 14. Setting the single-shot bit to ‘1’ enables the single-shot mode. After the single-shot has fired, it must be re-armed by setting this bit to ‘1’ again.

Target Peripheral Devices

DSP Interrupts

The ‘C6416 processor implements five interrupt input pins which allow external hardware events to directly trigger software or DMA activity. Processor interrupt inputs are supported on the baseboard through a set of control registers and multiplexers which allows application software to dynamically select the source of the signal which will drive each particular interrupt input as well as the polarity and edge or level sensitivity. Additionally, the interrupt controller in the support logic allows interrupt sharing by peripherals on the baseboard.

Software Support

The selection of DSP interrupts in the Pismo Library is controlled internally by the Device Drivers for the hardware. In ordinary circumstances these do not need to be manipulated by the user.

Hardware Implementation

The following table shows the addresses of the control registers for each processor interrupt input. A value written to the appropriate control register causes the interrupt mux to select the interrupt source given in the next table (see below).

Table 45. External Interrupt Input Control Register Addresses

Function	Address
NMI Interrupt Input Select	0x640D0000
External Interrupt Input 4 Select	0x64090000
External Interrupt Input 5 Select	0x640A0000
External Interrupt Input 6 Select	0x640B0000
External Interrupt Input 7 Select	0x640C0000

Table 46. External Interrupt Input Control Register Definitions

Bit	Interrupt Source
0	PCI Write FIFO Level
1	PCI Read FIFO Level
2	FIFOPort Transmit Level
3	FIFOPort Receive Level
4	PCI Mailbox Transmit
5	PCI Mailbox Receive
6	A/D FIFO Level

Bit	Interrupt Source
7	D/A FIFO Level
8	DDS
9	Event FIFO Level
10	Not used
11	Enable Interrupt (NMI only)
12	Acknowledged mode (NMI only) 0 = non-ack'd mode (default) 1 = ack'd mode
13	External Interrupt Input to J2 (SMB connector)
14..31	Not Used

The selection bits in each interrupt control register are identical on all interrupts, except for special fea- tures on NMI that are for sharing as noted. Set the corresponding bit true for each interrupt source that is used by that DSP interrupt. For example, if PCI Write FIFO Level is the interrupt source for the DSP external interrupt 4, then a 0x1 must be written to the DSP interrupt 4 select register at 0x64090000. Multiple interrupts sources may be enabled for NMI when it is used as a shared interrupt. Dedicated interrupts, INT4-7 should enable only one interrupt source. At reset, no interrupt sources are enabled.

Bit 12 is the mode selection for NMI for shared or dedicated mode described in the section on shared interrupts.

Conditioning for Interrupt Input Signals

Each interrupt source has polarity and edge/level selection so that nearly any interrupt source can be used by the DSP interrupts. The interrupt polarity is controlled by the interrupt source polarity register at 0x64110000 with each interrupt source, as numbered in the interrupt selection register above, con- trolled by a bit in the register. Edge/level selection, as defined in the interrupt type selection register at 0x64100000, allows the DSP to use either the interrupt source edge or level as the trigger condition for the interrupt. Polarity selection is normally used to control the edge used, rising or falling, or the level, requires that the DSP is always programmed for rising edge interrupts.

Table 47. Interrupt polarity and Type selection (polarity = 0x64110000, type = 0x64100000)

Interrupt Type	Polarity	Edge/Level
Rising edge	1	0
Falling edge	0	0
High level (1 = interrupt)	1	1
Low level (0 = interrupt)	0	1

Interrupts must remain active at least 15 ns after changing states, in either edge or level mode.

For edge detection, the logic samples the interrupt at 100 MHz, and watches for an edge. This edge detection requires that the interrupt be low for a minimum of 21 ns, then high for a minimum of 21 ns, for a valid edge to be detected.

Shared/ Dedicated Interrupts

The baseboard has a shared interrupt mode that may be used with the NMI only. Since the DSP has only five interrupts externally, many applications need a method for sharing interrupts to support all the peripheral devices on the baseboard efficiently. Shared mode has been developed so that multiple inter- rupt sources may share a single interrupt to the DSP.

In dedicated mode, each interrupt to the processor is steered directly from the selection matrix, through the edge/level and polarity conditioning directly to the processor. This allows the interrupt source to directly connect to the DSP. Dedicated interrupts are not shared amongst interrupt sources and should be used for the devices requiring the highest rates of interrupt servicing. These devices might be, for exam- ple, FIFO level interrupts from the PCI that need to be high speed. The dedicated mode does not have the burden of acknowledging the interrupts consumed so this mode is faster at interrupt servicing than the shared interrupts.

Shared interrupts allow multiple devices to share an interrupt to the processor. When the DSP receives a shared interrupt, it must read the interrupt status register associated with that interrupt to determine the interrupt source(s) requiring service. The DSP interrupt handling in this case should be capable of handling all the devices sharing this interrupt either alone of simultaneously to support the interrupt sharing. Upon completing the interrupt servicing, it is required that the DSP acknowledge the interrupt sources that were serviced. This prevents interrupts from being lost in the event that another interrupt source requires service in the meantime. The interrupt status/acknowledge registers are located at the address in the following table. Writing a '1' to any of the bits indicates that the interrupt has been serviced.

The bits have the same order as the interrupt source selection bits previously shown above.

Table 48. External Interrupt Status and Acknowledge Register Addresses

Interrupt	Address
NMI	0x64080000

All interrupt control registers default to 0 on power-up or board reset.

Note: The NMI interrupt is restricted to shared use only, and is currently used in polled mode exclu- sively. Restrictions in DSP BIOS limit the usefulness of the interrupt, so that no actual interrupt to the processor is currently enabled.

Note: The processor interrupt signals generated by the logic are active high (rising edge trigger), and the 'C6416 interrupt polarity control register must be programmed to the value 0x0 to correctly receive interrupts.

External User Interrupt Usage
The baseboard supports an external user interrupt. The external interrupt is available on JP4, an SMB connector. This input may be TTL or 50 ohm terminated as selected by jumper JP6 (jumper on = 50 ohm input).

Digital I/O

Quixote provides 32 bits of bidirectional digital I/O. The digital I/O port allows the baseboard to exchange digital handshaking and information signals with other hardware, control and signal other devices, and may be used for software troubleshooting tasks as well. The user DIO (UD) port that has separate control and data registers that allows byte-wide control of the direction. The UD digital IO port is on connector JP5 (MDR50 connector). See the appendix for the connector pinouts.

Software Support
The digital I/O hardware is controlled by the DigitalIo class. Its properties:

Table 49. DigitalIo Class Operations

Function	Type	Description
ClearBit()	Member Ftn.	Sets indicated bit to 0.
Config()	Member Ftn.	Configures bits to input or output
Data()	Property	Broadside Read/Write to DIO.
ReadBit()	Member Ftn.	Reads state of indicated bit
SetBit()	Member Ftn.	Sets indicated bit to 1.
WriteBit()	Member Ftn.	Writes value to indicated bit.

Hardware Implementation

Digital I/O port activity is controlled by the digital I/O configuration control and data registers, one for each port. Port direction is controlled by the configuration control register. The following diagrams gives the register definitions.

Bit Number:	31-5	4	3	2	1	0
Bit Field:	Reserved	External digital clock	DIR byte 3	DIR byte 2	DIR byte 1	DIR byte 0

Figure 57. UD Digital I/O Configuration Control Register (0x80300000)

Table 50. UD Digital I/O Configuration Control Register Definition (0x80300000)

Bit Field Name	Function
DIR byte 0	Direction control for digital I/O bits 7-0 (1 = output, 0 = input, default is input)
DIR byte 1	Direction control for digital I/O bits 15-8 (1 = output, 0 = input, default is input)
DIR byte 2	Direction control for digital I/O bits 23-16 (1 = output, 0 = input, default is input)
DIR byte 3	Direction control for digital I/O bits 31-24 (1 = output, 0 = input, default is input)
Ext Digital Clock	Enable the external digital clock. Default is internal clock.

Table 51. Digital IO Port Addresses

Port	Address
UD DIO	0x80380000

Data may be written to/read from the UD digital I/O port using the UD DIO port data register. Data written to ports bits which are set for output mode will be latched and driven to the corresponding port pins, while data written to input bits will be ignored. The input UD DIO may be clocked externally by enabling the external digital clock bit in the appropriate configuration register. If the internal clock is used, the data is latched at the beginning of any read from the port. Data read from output bits is equal to the last latched bit values (i.e. the last data written to the port by the DSP).

Digital I/O port pins are pulled up to 3.3V with ~10K ohm resistors in the logic device. Each DIO bit is ESD and overrange protected.

The UD port bits should be limited to a voltage range between 0 to 3.3V referenced to ground on the UD IO connector. Exceeding these limits may cause damage to the baseboard hardware.

Note! UD DIO pins are limited to 0 to 3.3V.

Digital I/O Timing

The following diagram gives timing information for the digital I/O port when used in external readback clock mode (see above for details). This data is derived from device specifications and is not factory tested.

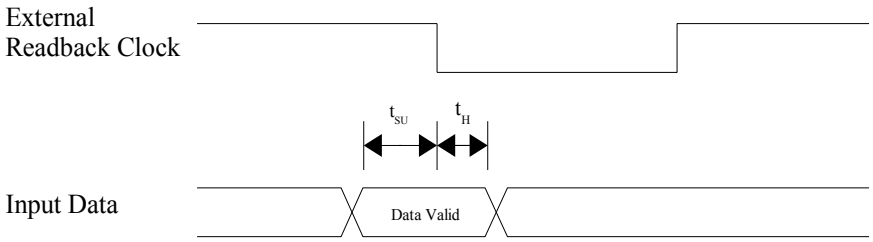


Figure 58. Digital I/O Port Timing

Table 52. Digital I/O Port Timing Parameters

Parameter	min. (ns)
t_{SU}	5
t_H	0

Digital IO Electrical Characteristics
The UD digital IO pins are LVTTL compatible, 3.3V tolerant pins driven by 3.3V logic. The logic chip used for the UD port is Xilinx XC2V2000/6000. The UD port output drive is +/-12mA, there is no series resistor. This allows the baseboard to drive LEDs, opto-couplers and relays directly under normal circumstances. Be aware that the logic chips are not heat sunk on the baseboard as delivered from the factory, and sourcing or sinking large amounts of current may result in power dissipation requiring heat sinking.

Caution! Forced air cooling or heat sinking may be required when using multiple digital IO pins to control heavy loads.

Timers

The Quixote provides a total of five counter/timers as well as the DDS used for timebase generation. These timers are independent and may be used as on board timebase generation for use in timing data acquisition, servo controls, real-time counters, and many other applications. The counter/timer functionality is divided into two devices: two 32-bit timer channels on the ‘C6416 processor and three 24-bit counter/timer channels in the logic. A 48-bit direct digital synthesizer (DDS)

channel in the AD9852 device is also provided. This section discusses the AD9852 synthesizer in detail: for more information on the on-chip timers, see the *TMS320C6000 Peripherals Reference Guide*.

Software Support

The software support for timers is handled in two ways. Since the most common use for a timer is to clock the Analog I/O, timer support is built into the Timebase objects used to configure the Analog In and Analog Out. If using timebase objects, there is no need to manage the clocks in the application as the object will manage the clock automatically.

For other uses, not involving Analog In or Analog Out, the Pismo library provides objects to manage the timers. These all compose a family of related classes to control timers..

Table 53. Timer Management Objects.

Object	Description
ClockBase	Base class for timer objects.
DdsClock	DDS Timer
DspClock	On-Chip Timers
ExtClock	Baseboard Timers

Hardware Implementation

On-chip Timers
The on-chip DSP timers are available for use as software timebases and interrupt generators. These timers may be driven by several sources as defined by software-controlled registers in the logic. The sources available to drive the DSP timers are listed in the following tables. DSP on-chip timer control register is located at 0x64140000.

Table 54. DSP On-chip Timer Clock Source Control Register (0x64140000)

Bit	Timer0 Input
0	DSP timer 0 input is DDS clock
1	DSP timer 0 input is SyncLink0
2	DSP timer 0 input is SyncLink1
3	DSP timer 0 input is ClockLink

Bit	Timer0 Input
4	DSP timer 0 us external clock
5..15	Not used
16	DSP timer 1 input is DDS clock
17	DSP timer 1 input is SyncLink0
18	DSP timer 1 input is SyncLink1
19	DSP timer 1 input is ClockLink
20	DSP timer 1 us external clock
21..31	Not used

AD9852 Direct Digital Synthesizer

The AD9852 direct digital synthesizer (DDS) is a precision programmable clock source which is capable of generating frequencies in the range of 0 to 105 MHz with a resolution of ~0.025 Hz/step. Unlike a digital counter-timer chip, which uses a digital counter to divide down a high input clock rate, the DDS uses phase-locked-loop synthesizer technology to tune a sine wave oscillator based on a 40-bit digital word. This method realizes a linear output frequency over input range rather than the nonlinear one associated with counter-timer chips, whose resolution drops dramatically as the period register used to program them falls. The DDS should be used when a precise and accurate clock is required by the application.

The AD9852 is mapped into memory as shown in the table below. The device is interfaced using the parallel I/O method, with one address to write data, one to trigger frequency/phase updates, and one to control the reset pin of the device.

Table 55. DDS Control Register (0x68010000)

Bit	Name	Function
0	DDS Reset	Reset DDS = '1' (default = '0')
1	DDS Frequency Update	Update the DDS frequency to the value in its data register when '1' (default = '0')
2	DDS Shape	Controls DDS function generation
3	DDS FSK	Controls FSK function in DDS
7	DDS Ext UD pin	Allows DDS update to be driven by DSP timer 0 output for frequency agile systems

The write clock address latches frequency/phase data into the AD9852 one byte at a time. The least significant eight bits of the processor bus carry the byte wide data. The frequency update address causes the output frequency and phase of the DDS clock to update to the values contained in its input latches. The new frequency will take effect when DDS Frequency UD is toggled.

The reset address causes an active high reset pulse to be generated to the AD9852. By default, the DDS reset is true at power-on or reset, so a '0' must be written to the DDS reset control bit in the control register before configuration and use. All three of these registers are write-only.

The output of the DDS may be used for a variety of functions including driving interrupts, as a timebase to the DSP or modules, or as an output. See the register descriptions for interrupt use, or timebase pin definition registers.

The Pismo Library includes support routines, which make it easy to set the AD9852's output frequency as discussed in the previous sections of this manual.

The DDS has very fine resolution allowing the application to tune the timebase to many frequencies. The absolute accuracy of the DDS timebase is approximately 200 ppm for room temperature applications. This absolute accuracy may vary with temperature and time. Calibration may be required in applications requiring higher precision timebases.

PXI Timing Support

Quixote supports multi-card synchronization and clock sharing through the PXI connections. Support for a Star Clock input/output and 5 PXI triggers allows the baseboard to send or receive clock and trigger signals to other cards In a PXI enabled chassis, Quixote can be integrated into complex multi-card systems with timebase and triggering synchronization for channel expansion and system coordination.

Software Support

The Pismo Library provides support for PXI as part of its Timebases. For a board to be configured as a Slave, select the SlaveTmb. Other timebases automatically configure themselves to be the PXI star or trigger master by default. See the description of the Timebase objects for more information on these objects.

Hardware Implementation

The SyncLink bus has one master who is the originator of all clock and trigger signals, while all other cards are referred to as slave devices. The slave cards receive clock/trigger signals from the master. Configuration of the SyncLink/ClockLink interfaces is controlled by the SyncLink/ClockLink control register.

Table 56. PXI Control Register Definition (0x80120000)

Bit	Function
0	PXI master enable (SyncLink master when 1, slave when 0)
1	PXI 0 set to DDS (when 1)
2	PXI 0 set to DSP timer 0 (when 1)
3	PXI 0 set to DSP timer 1 (when 1)
4	PXI 0 set to External clock (when 1)
5..16	Not used
17	PXI 1 set to DDS (when 1)
18	PXI 1 set to DSP timer 0 (when 1)
19	PXI 1 set to DSP timer 1 (when 1)
20	PXI 1 set to External clock (when 1)
20..25	Not used
26	PXI star output set to DDS (when 1)
27	PXI star output set to DSP timer 0 (when 1)
28	PXI star output set to DSP timer 1 (when 1)
29	PXI star output set to external clock (when 1)

Bit	Function
30..31	Not Used

When enabled as the master, Quixote has the following PXI functions.

Table 57. PXI Signals (when master)

PXI	Function
0	Programmable in PXI control register
1	Programmable in PXI control register
2	A/D start trigger
3	A/D stop trigger
4	D/A start trigger
5	D/A stop trigger

The control register defines whether the each PXI signal is a master or a slave, and what signals will be output on each of the PXI signals if the card is programmed as a master. Only one signal type can selected for each of the outputs.

Use of PXI triggers is recommended the trigger signals, and PXI star is recommended for the timebase signal.

Using PXI in slave mode, i.e. receiving timing/trigger signals for another card, is configured by selecting the appropriate PXI trigger in the timebase, trigger, or interrupt selection register.

External Clock Input

Quixote has an external clock input on J7, an SMB coaxial input connector. This clock is optionally 50 ohm terminated. The external clock may be used as a timebase source for the analog trigger controls, or in custom logic designs, offers a convenient input or output for any digital control signal or clock. This input feeds an analog comparator which uses ground as it's reference leg. Consequently, the external clock signal applied to this connector should be bipolar, swinging +/- 1.5 volts.

CAUTION: The external clock signal should be limited to +/-1.6V or damage may occur.

The trigger control mechanisms are configured by the Pismo toolset to allow use of the external clock input. Software functions provide setup and routing configuration to the logic.

Alert Log

Overview

Quixote has an “alert log” that can be used to monitor the data acquisition, playback, and other significant events. Using alerts, the application can know when triggers occurred, how much data was collected/played, be alerted when data is out-of-range or a data flow error occurred, and mark the data stream for other reasons. This provides a precision timed log of all of the important events that occurred during the acquisition and playback for interpretation and correlation to other system-level events.

Each time an enabled alert occurs, a message is put into the alert log telling the type of alert, when it occurred relative to the alert timestamp, and where it occurred in the data frame. The alert timestamp is a 1 us timer (32-bits wide) that all alerts use for their time reporting. Each alert message is put into the alert log FIFO where the DSP application can access them for data stream interpretation. Flow controls on the alert log FIFO allow the DSP application to control the data flow.

Software Support

Applications have different needs for alert processing. Aside from the bulk movement of data, most applications require some means of handling special conditions such as post-processing upon receipt of a stop trigger or closing a driver when an acquisition is completed.

When the alert system is enabled, the baseboard logic continuously monitors the status of the peripheral (usually analog) hardware present on the baseboard and submits a system generated alert into the alert queue whenever an alert condition is detected. It's also possible for application software to generate custom alert messages and submit them into the alert queue whenever the application deems it appropriate.

The Pismo library provides a built-in packet-based alert interface that can notify target applications asynchronously as exceptional stream conditions are detected by the hardware. A simple, bidirectional path for receiving and submitting alerts can be set up with minimal configuration for simple communication needs.

Setting up to receive alert messages is a simple, four-step process:

First, prototype the application’s alert message handler function, which you’ll be using to dispatch alert notifications from the system:

```
void  UnsolicitedAlertHandler( const AlertMessage & Msg );
```

Next, instantiate a binder function object of type AlertFtnType to bind your applications C function into an object which can be called by the alert software subsystem, when a hardware exception is detected:

```
FtnAlertHandler<AlertFtnType> UnsAlertHandler(UnsolicitedAlertHandler);
```

Now, implement your alert handler. This single function will be responsible for dispatching all received alert messages for your entire application. In the example below, each alert is simply inserted into an IIMessage object and sent to the Host PC for display, using the messaging subsystem.

```
//---------------------------------------------------------------------------
```

```
//  UnsolicitedAlertHandler() --  Process alerts
//---------------------------------------------------------------------------

void  UnsolicitedAlertHandler(const AlertMessage & Msg)
{
    IIMessage msg;
    msg.TypeCode(kEventReceivedMsg);
    msg.Data(0, Msg.Miscellaneous());
    msg.Data(1, Msg.Timestamp());
    msg.Data(2, Msg.FrameCount());
    Post(msg);
}
```

Finally, implement a function which will install the above alert handler, then enable the hardware alerts of interest. For example, the code below enables all analog input alerts, but no analog output alerts:

```
void  InitAlertTransport()
{
    //  Install handler for alerts
    InstallAlertHandler( &UnsAlertHandler );
    Alert.Reset();
    Alert.Enabled(aeAdcStart, true);
    Alert.Enabled(aeAdcStop, true);
    Alert.Enabled(aeAdcOutsideRange, true);
    Alert.Enabled(aeAdcFrameRollover, true);
    Alert.Enabled(aeAdcPretrigger, true);
    Alert.Enabled(aeUser, true);
}
```

Alert Objects

AlertMessage objects are packets of information that are transferred in the alert system. They contain functions to simplify the analysis of received alerts as well as the assembly of custom, user-specified alerts fit to be sent into the alert system by target application software.

For all practical purposes, you can think of alert messages as AlertMessage objects and ignore the details of the alert message format and transmission mechanics. Alert messages contain multiple fields of information including the time (AlertMessage::Time) and sample number (AlertMessage::FrameCount) at which the event occurred, the type of alert (AlertMessage::Type) and the peripheral (analog) channel associated with the alert (AlertMessage::Channel). For system-generated alerts, each of these fields will be filled by the alert hardware subsystem. However, for user generated alerts only the Alert- Message::UserData field may be modified by target application software.

On the host side, a similar class (AlertMessage) is used for managing alerts which are communicated to the Host PC via the message system.

Alert Binder Templates

Reception of an alert in Pismo is rather like an interrupt -- messages may come in from the alert subsystem at any time and the receiver must be notified when a message arrives. In a fashion similar to the interrupts, the messaging system provides a means to bind a callback function or method so that it can be called by the alert system when an alert is detected.

The Binder system can be thought of as a more flexible and powerful version of a function pointer vari- able, allowing a user callback function to be called. Since the binder objects are C++ templates, member functions can be bound to an interrupt, which a callback function can never do. One simplification of the message binder is that the type of callback is fixed -- the argument of the callback function is always a reference to an AlertMessage object.

The Binder system is powerful, yet in practice is quite simple to use. This system illustrates the power of the C++ language to contain a complicated system in a simple-to-use package.

Using Alerts in Application Code
When a message is recieved by the Alert system, the alert is dispatched to a single application handler for 'Unsolicited Alerts'. A handler for these messages needs to be installed by a call to the function InstallAlertHandler() before messages can be received:

```
//  Install handler for un-channelized messages
InstallAlertHandler( &UnsAlertHandler );
```

A second global function, Post() will submit a custom, application-specific alert to the alert hardware subsystem. Posted alerts generate alert callbacks just as hardware detected alerts do, so that application code can encode alert messages into the alert message stream as appropriate for the application. This is the signature for the Post() function.

```
bool  Post(AlertMessage & msg);
```

Host Side Support
The host-side support for alerts is analogous to the support on the target. There is a message class on the host, AlertMessage. It supplies properties and methods to access the fields and data of the message.

To communicate the contents of an alert to the Host, copy the fields of the AlertMessage into an IIMes- sage object, then use the messaging system to transport the alert to the Host PC application.

Hardware Implementation

The alert log can monitor many types of important events and generate alert messages for the DSP application that allow the application to monitor and control the data acquisition and playback. These alerts may be enabled in the alert log enable register as follows.

Table 58. Alert Log Enable Register (0x80060000)

Bit	Alert	Purpose
0	A/D start trigger	When and where in the data set an A/D start trigger occurred.
1	A/D stop trigger	When and where in the data set an A/D stop
2	A/D out-of-range	When and where in the data set an A/D out-of-range occurred. (The A/D sample = 0x7FFF or 0x8000).
3	A/D frame count rollover	When and where the frame counter rolled over from its maximum to zero. The software can use this to extend the counter.
4	A/D timebase event during pretrigger or trigger active	When and where in the data an A/D timebase trigger occurred.
5	Not used	

Bit	Alert	Purpose
6	A/D overflow	When and where an A/D overflow occurred, indicating at least one data set was lost.
7	A/D user event	The DSP can write into the log to mark points and times of important system-level events.
8..15	Not used	
16	D/A start trigger	When and where in the data set an A/D start
17	D/A stop trigger	When and where in the data set an A/D stop
18	Not used	
19	D/A frame counter rollover	When and where the frame counter rolled over from its maximum to zero. The software can use this to extend the counter.
20	D/A timebase event during pretrigger or trigger active	When and where in the data a D/A timebase trigger occurred.
21	D/A underflow	When and where a D/A underflow occurred, indicating at least one conversion was missed.
22	Alert Timestamp rollover	Indicates that the 32-bit timestamp for alert log rolled over from its maximum to zero. This can be used by software to extend the alert timebase.
23..31	Not used	

The alerts may be enabled by writing a ‘1’ to any of the bits, all are disabled by default.

The timebase events for the D/A and A/D allow the application to record the time a conversion trigger occurred during the active trigger region. This is useful for aperiodic or external timebases that the application does not have prior knowledge of the trigger timing. Enable this with care, since it generates a three DWORD message for each data set, and may overwhelm the log at high data rates.

An out-of-range alert is generated when an enabled channel, after error compensation, is equal to either 0x3FFF or 0xC000, which indicates a full scale reading. The alert will become true for out-of-range under this full scale condition. This does not stop data acquisition, it only produces an out-of-range alert if so enabled. For the out-of-range alert, once this condition is seen, it remains true until the A/D run bit is false, indicating that the A/D engine has been turned off (reset). Only one event will be produced for the out-of-range condition, which will have the offending channel number encoded on the first word of the three word message.

Alert Log Message Format

The alert log messages are entered into the log as a set of three 32-bit words, with the following format.

Table 59. Alert Message Format (0x805E0000)

Word	Bits	Description

0	31..27	Alert type
	26..24	“000”
	23..16	error channel (valid only for out-of-range alerts)
	15..0	0x0000
1	31..0	timestamp
2	31..0	frame count or user alert data

The first word contains the alert type, as encoded in the following table, and the error channel for out-of-range alerts (invalid for all other alert types). The second word is the 32-bit timestamp. The final word is either the frame count, or in the special case of a user alert it is a 32-bit data word.

Table 60. Alert Log Event Types

Alert Type	Description
0b10000	A/D start trigger
0b10001	A/D stop trigger
0b10010	A/D out-of-range
0b10011	A/D frame count rollover
0b10100	A/D Time base trigger during active trigger or pretrigger
0b10110	A/D overrun error
0b00000	D/A start trigger
0b00001	D/A stop trigger
0b00011	D/A frame count rollover
0b00100	D/A Timebase trigger during active trigger or pretrigger
0b00110	D/A overrun error
0b00111	D/A user event
0b10111	Timestamp rollover

Controlling and Monitoring the Alert Log

The Alert Log reports the FIFO not empty status on bit 14 of the Status register (0x80580000). When true (‘1’), this bit tells whether the Alert FIFO has any data. This may also be enabled as an interrupt to the processor in the interrupt selection register.

The Alert FIFO may be reset in the control register (0x80320000), bit 3. When true (‘1’), this resets the event FIFO.

Alert Log Data Flow

Writing user alerts to the Alert Log requires that the application monitor the Alert Log Busy bit in the status register (0x80580000, bit 15). When true (‘1’), this means a user alert is already pending, and no more user alerts should be written until this bit is false. Do not DMA to the event log, as the DMA con- troller has no means to respect the user alert busy bit. Data is written to the Alert Log at 0x805E00000.

The Alert Log FIFO has a not empty indicator that may be used interrupt the CPU or DMA to move the data from the Alert FIFO. Data is read from address 0x805E0000.

Serial EEPROM Interface

Quixote has a serial EEPROM for storing data such as board identification, calibration coefficients, and other data that needs to be stored permanently on the card. This memory is 4K bits in size. Functions for using the Serial EEPROM are included in the Pismo Toolset that allow the software application programmer to easily write and read from the memory without controlling the low-level interface.

As delivered from the factory, this EEPROM contains the calibration coefficients used for the A/D and D/A error correction.

StarFabric

StarFabric provides a high performance data link for intercard communcation and system expansion. Quixote provides dual 2.5 Gbps, full-duplex StarFabric ports compliant to PCIMG 2.17 standard. These StarFabric ports may be configured in a variety of ways to allow multiple cards, such as Quixote or Quadia DSP cards, to communicate over dedicated high performance links, or allow Quixote to be inte- grated into a larger StarFabric-enabled system.

StarFabric Modes

Quixote has a StarFabric interface on the local PCI bus that allows the PCI devices on Quixote to be integrated into a larger PCI system. The StarFabric controller is the StarGen SG2010. The SG2010 is a PCI to StarFabric bridge device and has several modes of operation that allow it to operate as a “root” or “leaf” device on the StarFabric, and also whether or not PCI bridge capability is enabled.

<< more on this to be added>>

Hardware Interface

Quixote implements a PICMG 2.17 compliant StarFabric interface using compact PCI connector J3. Each StarFabric Link is 4 differential signal pairs in each direction. Mapping to the J3 connector is shown in the Connector appendix.

<< more on this to be added>>

Software Support

<< more on this to be added>>

PMC Module Support (not released)

Quixote supports one PMC module site for IO expansion as either a PCI device, or using the J4 connec- tor for a private data interface to the Virtex2 logic for custom logic developments. A wide variety of modules are available from many vendors to support many types of IO requirements.

PMC PCI Interface

This PMC module site connects to the local PCI bus as 64-bit device. The local PCI bus supports 33 MHz and 3.3V signaling levels. Busmastering is supported.

The four interrupts from the PMC module may be routed to the PCUI bus, or to the DSP as required to support the PMC IO requirement.

<< more description will be added when this feature is released>>

PMC J4 Support

The PMC module site J4 connector can be used as a private data interface to the FPGA for custom logic designs. The J4 connector provides 64 connections to the FPGA, that may be used either as LVTTL connections, or as 31 pairs of LVDS signals (2 signals must be LVTTL only). This allows the FPGA to be integrated into the data path of the PMC devices if high speed processing, formatting or analysis is required.

Connector Pinouts and Physical Information

Note : For Rev A, see drawing for connector numbers. Tables below are Rev B.

Analog IO Connectors

Connector type: SMB 50 Ohm

Mating connector: AMP P/N 413985-3 (straight) or
AMP P/N 414002-7 (right angle)

Table 61. Analog IO Connectors (Rev B)

Function	Connector
A/D input 0	J1
A/D input 1	J3
D/A output 0	J5
D/A output 1	J6
Analog Comparator Input	J2
Ext Int	J4

JP5 – UD Digital I/O Connector

Connector types: Miniature D-shell, spring leaf connector (MDR)

Number of pins: 50

Mating connector: 3M part number 10150-6000EC or
Tyco (Amp) 2-175677-7

The following table gives the pin numbers and functions for the JP5 connector.

Table 62. UD Digital I/O Connector Pinout

Pin Number	JP5 Function	Direction (from Quixote)	Virtex2 FPGA Signal Name	Virtex2 Pad Number	CCHICO PLUS Pin Number
1	Digital I/O bit 0	I/O	UD<0>	K2	50
26	Digital I/O bit 1	I/O	UD<1>	L2	49

Pin Number	JP5 Function	Direction (from Quixote)	Virtex2 FPGA Signal Name	Virtex2 Pad Number	CCHICO PLUS Pin Number
2	Digital I/O bit 2	I/O	UD<2>	K5	48
27	Digital I/O bit 3	I/O	UD<3>	L5	47
3	Digital I/O bit 4	I/O	UD<4>	M6	46
28	Digital I/O bit 5	I/O	UD<5>	L6	45
4	Digital I/O bit 6	I/O	UD<6>	J1	44
29	Digital I/O bit 7	I/O	UD<7>	K1	43
5	Digital I/O bit 8	I/O	UD<8>	K4	42
30	Digital I/O bit 9	I/O	UD<9>	J4	41
6	Digital I/O bit 10	I/O	UD<10>	L7	40
31	Digital I/O bit 11	I/O	UD<11>	K7	39
7	Digital I/O bit 12	I/O	UD<12>	H3	38
32	Digital I/O bit 13	I/O	UD<13>	J3	37
8	Digital I/O bit 14	I/O	UD<14>	H5	36
33	Digital I/O bit 15	I/O	UD<15>	J5	35
9	Digital I/O bit 16	I/O	UD<16>	K6	34
34	Digital I/O bit 17	I/O	UD<17>	J6	33
10	Digital I/O bit 18	I/O	UD<18>	H2	32
35	Digital I/O bit 19	I/O	UD<19>	J2	31
11	Digital I/O bit 20	I/O	UD<20>	H4	30
36	Digital I/O bit 21	I/O	UD<21>	G4	29
12	Digital I/O bit 22	I/O	UD<22>	L8	28
37	Digital I/O bit 23	I/O	UD<23>	K8	27
13	Digital I/O bit 24	I/O	UD<24>	F1	26
38	Digital I/O bit 25	I/O	UD<25>	G1	25
14	Digital I/O bit 26	I/O	UD<26>	F3	24
39	Digital I/O bit 27	I/O	UD<27>	G3	23
15	Digital I/O bit 28	I/O	UD<28>	J7	22
40	Digital I/O bit 29	I/O	UD<29>	J8	21
16	Digital I/O bit 30	I/O	UD<30>	F2	20
41	Digital I/O bit 31	I/O	UD<31>	G2	19
17	Digital I/O bit 32	I/O	UD<32>	G5	18
42	Digital I/O bit 33	I/O	UD<33>	F5	17
18	Digital I/O bit 34	I/O	UD<34>	H6	16
43	Digital I/O bit 35	I/O	UD<35>	G6	15
19	Digital I/O bit 36	I/O	UD<36>	D1	14
44	Digital I/O bit 37	I/O	UD<37>	E1	13
20	Digital I/O bit 38	I/O	UD<38>	E4	12
45	Digital I/O bit 39	I/O	UD<39>	F4	11
21	DSP IO bit GP0	I/O			10

Pin Number	JP5 Function	Direction (from Quixote)	Virtex2 FPGA Signal Name	Virtex2 Pad Number	CCHICO PLUS Pin Number
46	DSP IO bit GP3	I/O			9
22	DSP timer TIN2	Input			8
47	DSP timer TOUT2	Output			7
23	External ADC stop trigger	Input			6
48	External ADC start trigger	Input			5
24	External DAC stop trigger	Input			4
49	External DAC start trigger	Input			3
25	+3.3V	Power			2
50	DGND	Power Return			1

The following figure gives the pin orientation for the UD digital I/O connector.

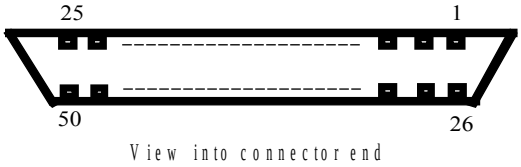


Figure 59. Digital I/O Connector Pin Orientation (JP5)

JN4 - PMC Private IO Connector (Rev B or higher only)

The PMC JN4 connector is used to interface PMC modules directly to the FPGA as a private data path.

Connector types: 1 mm double row, IEEE 1386 compatible vertical connector
Number of pins: 64
Mating connector: Molex P/N 71436

The following table gives the pin numbers and functions for the PMC JN4 connector ***on the Rev B cards.***

Table 63. PMC JN4 Pinout for Rev B

PMC IO2	1	I/O	I/O	2	PMC IO1
PMC IO4	3	I/O	I/O	4	PMC IO3
PMC IO6	5	I/O	I/O	6	PMC IO5
....					
PMC IO<pin+1>	Even pin	I/O	I/O	Odd pin	PMC IO<pin-1>
...					...
PMC IO62	61	I/O	I/O	62	PMC IO61
PMC IO64	63	I/O	I/O	64	PMC IO63

The following table gives the pin numbers and functions for the PMC JN4 connector ***on the Rev C and higher cards.***

Table 64. PMC JN4 Pinout for Rev C and Higher

PMC IO1	1	I/O	I/O	2	PMC IO2
PMC IO3	3	I/O	I/O	4	PMC IO4
....					
PMC IO<pin>	Even pin	I/O	I/O	Odd pin	PMC IO<pin>
...					...
PMC IO61	61	I/O	I/O	62	PMC IO62
PMC IO63	63	I/O	I/O	64	PMC IO64

The PMC IO pins may be used as LVTTL (3.3V ***only***) or as LVDS pairs. LVDS pairs are 2-3, 3-4....63- 64, pins 0 and 1 are not usable as an LVDS pairs.

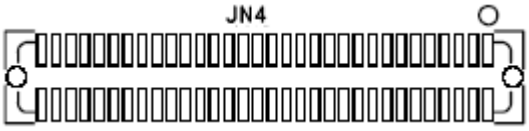


Figure 60. PMC JN4 Connector Pin out

This image of the solder mask of the JN4 connector gives the orientation of the pins on the connector.

JP2 – Virtex2 JTAG Connector

Connector type:	14 pin 2mm double row male header
Number of pins:	14
Mating connector:	AMP P/N 111623-3

The following table gives the pin numbers and functions for the JP2 connector.

Table 65. Virtex2 JTAG Connector Pinouts

Ground	1	P	P	2	3.3V
Ground	3	P	I	4	TMS
Ground	5	P	I	6	TCK
Ground	7	P	O	8	TDO
Ground	9	P	I	10	TDI
Ground	11	P		12	No Connect
Ground	13	P		14	No Connect

JP7 – DSP JTAG Connector

Connector types:	14 pin 0.1” double row shrouded male header center polarized
Number of pins:	14
Mating connector:	AMP P/N 746285-2

The following table gives the pin numbers and functions for the JP7 connector.

Table 66. JP7 DSP JTAG Connector Pinouts

TMS	1	I	I	2	TRST*
TDI	3	I	P	4	Digital Ground
3.3V	5	P	P	6	Digital Ground
TDO	7	O	P	8	Digital Ground
TCK	9	I	P	10	Digital Ground
TCK	11	I	P	12	Digital Ground
EMU0	13	I	I	14	EMU1

Board Layout and Connectors

A dimensional schematic and the board layout is displayed below. Please review these drawings to familiarize yourself with the circuit board's configuration and connections

cQuixote Revision A Connectors

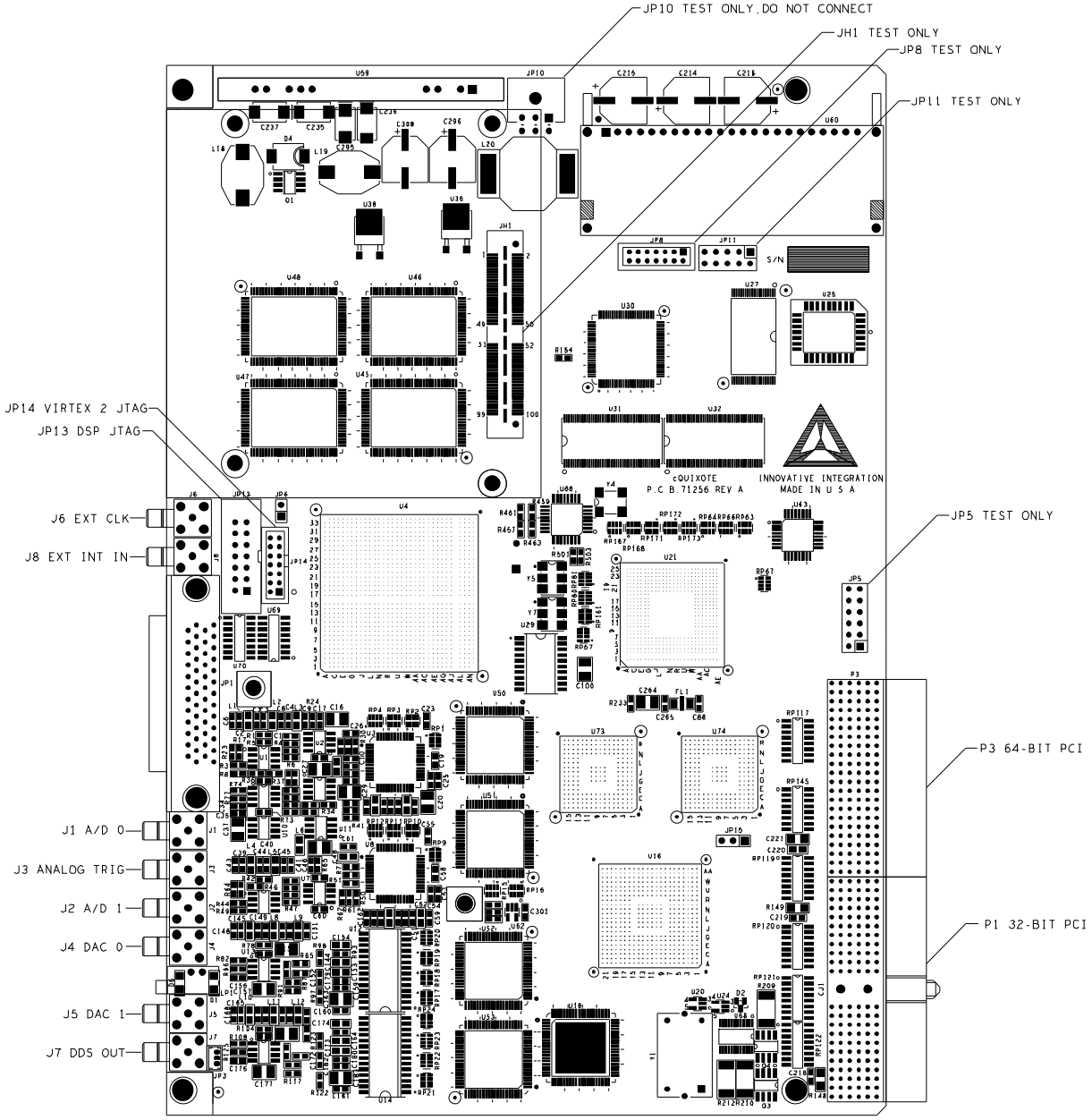


Figure 61. cQuixote Rev. A Connectors

cQuixote Revision B Connectors

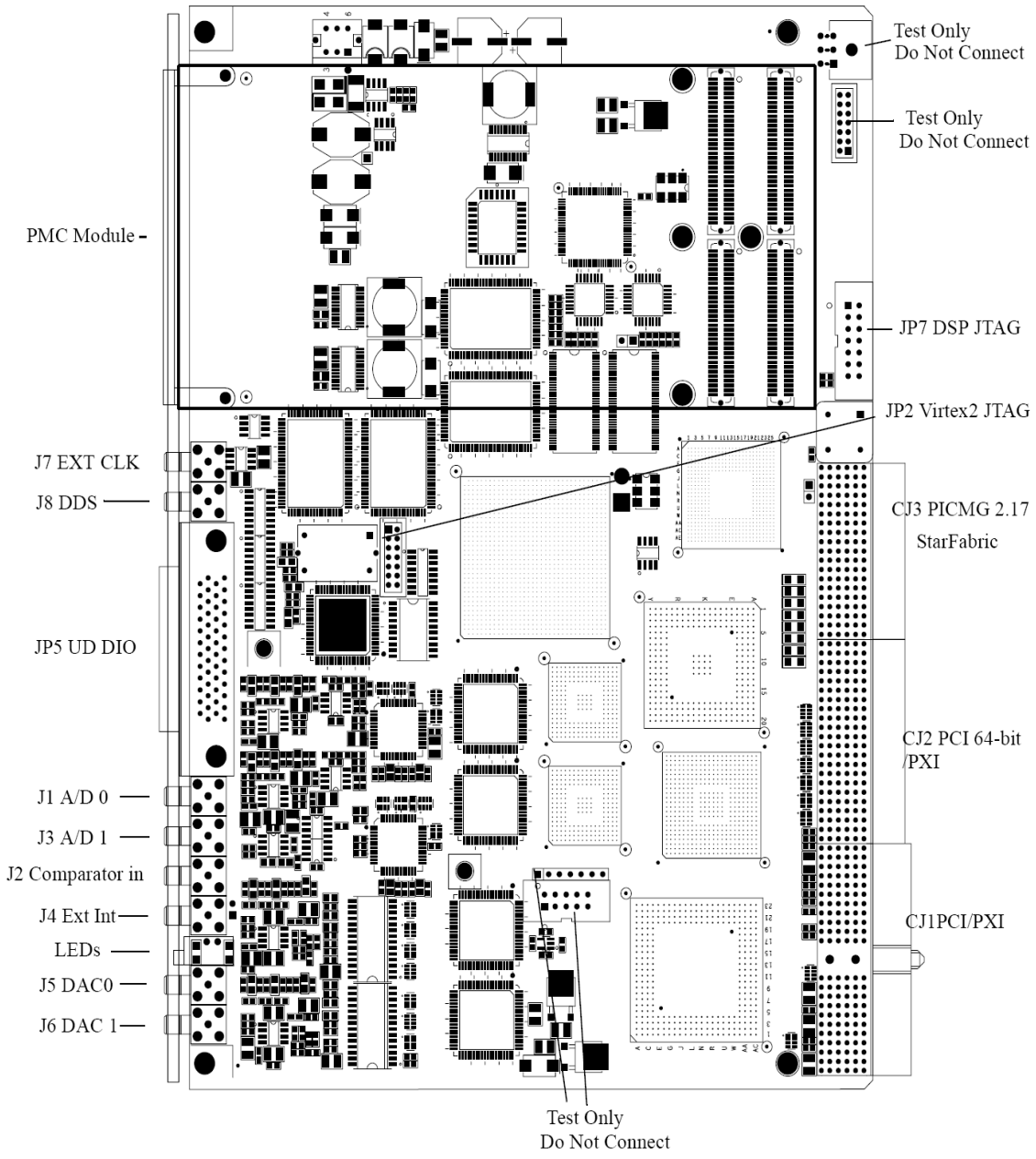


Figure 62. cQuixote Rev. B Connectors

cQuixote Revision C and Revision D Connectors

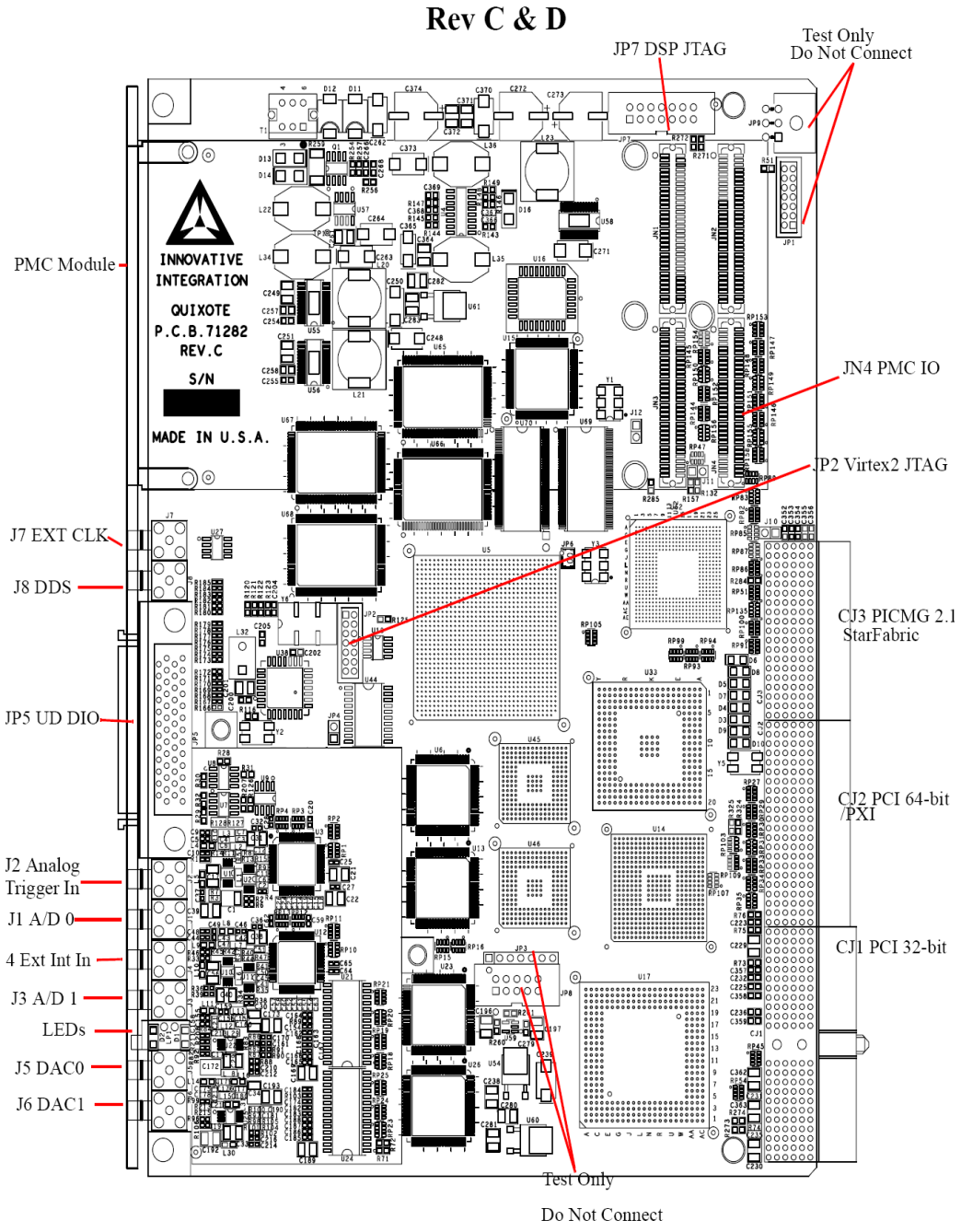


Figure 63. cQuixote Rev. C and D Connectors

cQuixote Revision E Connectors